Universität Paderborn
Fachbereich 17 · Informatik

Prozessorientierte Integration von Softwarekomponenten durch XML-basierte Workflow-Modelle

Diplomarbeit für den
integrierten Studiengang Informatik
nach Diplomprüfungsordung 4

von Björn Lütkemeier und Sebastian Thöne

vorgelegt bei Prof. Dr. Gregor Engels
Lehrstuhl für Datenbank- und Informationssysteme

Paderborn, 5. Dezember 2001

Danksagung

Diese Diplomarbeit entstand im Rahmen unseres Informatikstudiums an der Universität Paderborn. Sie wurde durch Prof. Dr. Gregor Engels und Dipl.-Inform. Dipl.-Phys. Ralph Depke betreut. Für ihre Hilfen und Ratschläge bei der Erstellung der Arbeit sind wir sehr dankbar. Besonderen Dank sagen wir auch der S&N AG in Paderborn, da diese Arbeit in enger Kooperation mit dem Unternehmen entstanden ist. Für die betreuende Hilfestellung, insbesondere durch Matthew Langham, sowie die bereitgestellten Hilfsmittel und Arbeitsplätze bedanken wir uns ganz herzlich. Außerdem danken wir der Deutschen Bausparkasse Badenia AG, die uns freundlicherweise Dokumentationen und Quellcodes zu ihrem E-Business-Projekt zur Verfügung gestellt hat. Allen anderen, die zum Erfolg dieser Diplomarbeit beigetragen haben, gilt ebenfalls unser herzlicher Dank.

Paderborn, im Dezember 2001 Björn Lütkemeier & Sebastian Thöne

Erklärungen

Ich versichere, dass ich die kenntlich gemachten Teile dieser Diplomarbeit, die als Gruppenarbeit mit Sebastian Thöne entstanden ist, selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, habe ich unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Paderborn, Dezember 2001

Björn Lütkemeier

Ich versichere, dass ich die kenntlich gemachten Teile dieser Diplomarbeit, die als Gruppenarbeit mit Björn Lütkemeier entstanden ist, selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, habe ich unter genauer Angabe der Quelle deutlich als Entlehnung kenntlich gemacht.

Paderborn, Dezember 2001

Sebastian Thöne

Inhaltsverzeichnis

1 Einleitung 5

2 Grundlagen des Workflow-Managements 9
2.1 Prozessorientierte Integration im E-Business 9
2.2 Prozesse in einem Unternehmen 11
2.2.1 Der Begriff „Workflow“ 12
2.2.2 Klassifizierung von Informationsprozessen 14
2.2.3 Integration existierender Anwendungen 15
2.3 Modellierung von Prozessen 17
2.3.1 Organisationsmodellierung und statische Modelle 17
2.3.2 Vorgangsmodellierung und dynamische Modelle 18
2.3.3 Analyse und Simulation der Modelle 18
2.4 Unterstützung durch Software 19

3 Anforderungsanalyse 23
3.1 Fallstudie: E-Business-Plattform eines großen Finanzunternehmens 23
3.1.1 Projektziele 23
3.1.2 Anforderungen 24
3.1.3 Architekturvorschlag 26
3.2 Anforderungen an Workflow-Systeme im E-Business 28
3.3 Anforderungen an Workflow-Modellierungssprachen 31
3.4 Zielsetzungen dieser Arbeit 33

4 Evaluierung vorhandener Workflow-Systeme 35
4.1 Microsoft BizTalk Server 2000 35
4.1.1 Systemarchitektur von BizTalk Server 2000 37
4.1.2 Komponentenintegration mit BizTalk Server 2000 38
4.1.3 Workflow-Modellierung mit BizTalk Server 2000 38
4.1.4 Workflow-Ausführung mit BizTalk Server 2000 39
4.2 Komponentenkonfiguration mit WARP 40
4.2.1 Systemarchitektur von WARP 40
4.2.2 Komponentenintegration mit WARP 41
4.2.3 Workflow-Modellierung mit WARP 42
4.2.4 Workflow-Ausführung mit WARP 42
4.3 E-Business-Plattform sunShine 43
4.3.1 Systemarchitektur von sunShine 44
4.3.2 Workflow-Ausführung mit sunShine 45
4.3.3 Komponentenintegration mit sunShine 46
4.4 Bewertung 48

5 Evaluierung vorhandener Workflow- Modellierungssprachen 51
5.1 Petri-Netze 51
5.1.1 Klassische Petri-Netze 51
5.1.2 High-Level-Petri-Netze 52
5.1.3 Geschäftsprozessmodellierung mit Petri-Netzen 53
5.1.4 Eignung für Prozessdiagramme 55
5.1.5 Datenfluss in Petri-Netzen 58
5.2 Ereignisgesteuerte Prozessketten 59
5.2.1 Basiskonstrukte des EPK-Modells 59
5.2.2 Formale Beschreibung der Syntax 61
5.2.3 Geschäftsprozessmodellierung mit EPK 62
5.2.4 Eignung für Prozessdiagramme 63
5.3 UML-Aktivitätendiagramme 66
5.3.1 Basiskonstrukte von Aktivitätendiagrammen 66
5.3.2 Geschäftsprozessmodellierung mit UML 69
5.3.3 Eignung für Prozessdiagramme 70
5.4 Bewertung 71

6 XML-Prozesse und Prozessdiagramme 75
6.1 Konzeption von XML-Prozessen 75
6.1.1 Dokumentmigration 75
6.1.2 Komponentenintegration 79
6.1.3 Modell- und Ausführungsebene 81
6.1.4 Management von XML-Prozessen 82
6.2 Modellierung mit Prozessdiagrammen 84
6.2.1 Aktivitäten zur Komponentenintegration 87
6.2.2 Parametrisierung von Komponenten 87
6.2.3 Transitionen als XML-Dokumentfluss 88
6.2.4 Guard-Bedingungen 89
6.2.5 Behandlung von parallelen Teilprozessen 89
6.2.6 Einschränkungen gegenüber Aktivitätendiagrammen 90
6.3 Modellierung der Dokumenttypen 92
6.3.1 Das Port-Konzept zur Typisierung von Dokumenten 94
6.3.2 Wiederverwendbarkeit und Konsistenz durch typisierte Transitionen . 104
6.3.3 Formale Definition von Ports und typisierten Transitionen 109
6.4 Bewertung 119

7 Metamodell für Prozessdiagramme 121
7.1 Das UML-Metamodell 121
7.1.1 Architektur von UML 121
7.1.2 Metamodell für Aktivitätendiagramme 122
7.1.3 Object Constraint Language 123
7.2 UML-Erweiterungsmechanismen 126
7.2.1 Stereotypen 127
7.2.2 Tag Definitions und Tagged Values 128
4.2.3 Constraints 128
7.2.3 Profile 128
7.3 UML-Profil für Prozessdiagramme 129
7.3.1 Umsetzung des Port-Konzepts in UML-Datentypen 130
7.3.2 Definition von Stereotypen für Prozessdiagramme 132
7.3.3 Wohlgeformtheitsregeln 143
7.4 Implementierung des erweiterten Metamodells 145
7.5 Validierung von Prozessdiagrammen 154
7.6 Automatische Portberechnung 156

8 Beschreibung von Prozessen in XML 161
8.1 Anforderungen an Prozessbeschreibungssprachen 161
8.2 Vorhandene XML-Formate für Prozessmodelle 164
8.2.1 Beschreibung von Prozessen mit XLANG 164
8.2.2 Der XML Metadata Interchange Standard (XMI) 166
8.2.3 Bewertung 170
8.3 Process Markup Language (PML) 172
8.3.1 Beschreibung von Ports 172
8.3.2 Beschreibung des statischen Modells 176
8.3.3 Beschreibung von Prozessen 179
8.3.4 Bewertung 186
8.4 Werkzeuge für die PML-Verarbeitung 187
8.4.1 PML-Importer 187
8.4.2 PML-Exporter 188

9 Modellierungswerkzeug für Prozessdiagramme 191
9.1 Der Prozesseditor 191
9.2 Bedienung des Prozesseditors 192
9.2.1 Verwaltung in Projekten 193
9.2.2 Ansicht des Projektbaums 193
9.2.3 Eingabemasken 195
9.2.4 Eingabe von Ports 196
9.2.5 Konstruieren eines Prozesses 197
9.3 Implementierung des Prozesseditors 205
9.3.1 Klasse AppFrame 205
9.3.2 Anwendung des Model-View-Controller-Konzepts 206
9.3.3 Die Prozesszeichenfläche 208

10 Ausführung von XML-Prozessen 211
10.1 Semantik von Prozessdiagrammen 211
10.2 Der Prozessinterpreter 217
10.2.1 Interne Datenstruktur zum Zugriff auf Prozessmodelle 217
10.2.2 Ausführende Klassen 219
10.2.3 Synchronisationsmechanismus 222
10.2.4 Aufruf von Services 223
10.3 Fehlerbehandlung 225
10.3.1 Fehlertoleranz 226
10.3.2 Explizite Modellierung von Fehlerbehandlungen 227
10.4 Transaktionale Prozessausführung 228
10.4.1 Eigenschaften von Transaktionen 228
10.4.2 Transaktionen in XML-Prozessen 229
10.4.3 Fehler bei der Ausführung von Services 230
10.4.4 Ausfall des Prozessinterpreters 231
10.5 Entwicklung von Konnektoren 233
10.6 Integration in sunShine 235
10.6.1 Der Sunflow-Transformer 235
10.6.2 sunShine-Prozessumgebung 235

11 Evaluierung anhand einer Fallstudie 237
11.1 Vergleich der Lösungsansätze 237
11.2 Erfüllung der Anforderungen 243

12 Schlussbetrachtungen 245
12.1 Zusammenfassung 245
12.2 Ausblick 250

Literatur und Quellen 255

1 Einleitung

Ein Blick in die Historie des Internets zeigt, dass wir uns am Beginn einer neuen Periode befinden: Bislang war das Internet von der *Kommunikationsära* geprägt (vgl. [Mic00] S.2), der ersten Ära der Internetgeschichte. Sie legte den Grundstein für die überaus erfolgreiche Verbreitung der Internetkommunikation durch die Schaffung entsprechender Infrastrukturen und einheitlicher Protokolle, die auf offenen Standards basieren. Dies war der erste notwendige Schritt zur Etablierung der elektronischen Geschäftsabwicklung (E-Business). Doch diese Infrastruktur allein genügte noch nicht zur Umsetzung vollständiger elektronischer Geschäftsbeziehungen. Vielmehr waren zunächst E-Mail-Kommunikation und Datenpräsentation auf Websites die hauptsächliche Anwendung der neuen Kommunikationsinfrastruktur. Dies eröffnete den Benutzern in den ersten Jahren eine neue und bisher nicht da gewesene Möglichkeit zum Informationsaustausch. Aber die neuen Technologien erreichten schnell auch ihre Grenzen: Die Beschreibungssprache *HyperText Markup Language* (HTML, siehe [HTML99]) dient in erster Linie zur Formatierung und Präsentation von Texten und Grafiken, die elektronische Post (E-Mail) zum einfachen Versenden von Textpaketen und Dateien. Keine dieser beiden Technologien ermöglicht es jedoch den Geschäftspartnern, strukturierte Daten, zum Beispiel über Auftrags- und Rechnungseingang, miteinander auszutauschen, so dass sie direkt von den eigenen Anwendungssystemen verarbeitet werden könnten. Nur der Ansatz des elektronischen Datenaustausches über EDI (Electronic Data Interchange) verfolgt dieses Ziel, wird aber durch proprietäre und zweckgebundene Netzwerke und Protokolle realisiert (vgl. [Mic00] S.2), was sich nachteilig auf die Flexibilität und Offenheit der Geschäftsbeziehungen auswirkt.

In der zweiten Ära des Internets, der *Integrationsära*, vollzieht sich seit Ende der neunziger Jahre ein allmählicher Wandel von der einfachen Kommunikation zwischen Geschäftspartnern hin zur Integration ihrer Anwendungssysteme und Geschäftsprozesse. Mit dem Begriff *Integration* ist hier die Kopplung der meist heterogenen Teilsysteme eines Unternehmens gemeint, die oft als Insellösungen unabhängig voneinander und nebeneinander existieren. Damit diese Systemkomponenten Daten und Informationen austauschen können, sind entsprechende Schnittstellen und Protokolle nötig. So wie bei der betriebswirtschaftlichen Geschäftsprozessoptimierung die Unternehmensproduktivität durch ein verbessertes Zusammenspiel der einzelnen Tätigkeiten und Verarbeitungsschritte im Unternehmen gesteigert wird, ist dies auch durch die Integration der Softwaresysteme möglich. Ein zweiter, wichtiger Aspekt von Softwareintegration im Zeitalter des E-Business ist die Anbindung der vorhandenen Systeme an das Internet, so dass sie auch über das Netz von Kunden, Mitarbeitern und Partnern aufgerufen werden können. Diese Integrationsschritte bilden die Voraussetzung für eine unternehmensübergreifende Integration, bei der die Softwaresysteme mehrerer Unternehmen, zum Beispiel der Beteiligten in einer Lieferkette, miteinander vernetzt werden.

Grundlage für diese Entwicklung von der Kommunikation zur Integration ist neben der bereits bestehenden Kommunikationsinfrastruktur die Entwicklung geeigneter Technologien zum Austausch strukturierter Daten, durch die sich die verschiedenen Anwendungssysteme miteinander koppeln lassen. Einen starken Schub bei der Umsetzung dieser Ziele brachte die Einführung der Extensible Markup Language (XML, siehe

[XML00]) Ende der neunziger Jahre. XML stellt seitdem quasi das Alphabet beim Datenaustausch im E-Business dar (vgl. [Mic00] S.2f). Da XML jedoch nur eine Technik ist, um die unterschiedlichsten Datenformate individuell zu definieren, bedarf es für den Austausch von XML-Dokumenten über lokale Grenzen eines Unternehmens hinweg der Abstimmung gemeinsamer XML-Formate für die speziellen Anwendungszwecke. Dazu müssen die genauen Strukturen der XML-Formate durch Dokumenttypdefinitionen und Schemata festgelegt werden. Da nach der Verabschiedung des XML-Standards nun der Einigung auf solche anwendungsspezifischen XML-Formate eine essentielle Bedeutung zukommt, werden in dieser Richtung intensive Angleichungsbestrebungen durchgeführt: Standards wie die *Electronic Business XML* (ebXML, siehe [EbX01]) definieren genaue Schemata und Regeln zur Erstellung von XML-Dokumenten mit dem Ziel, die beim elektronischen Geschäftsverkehr benutzten XML-Formate zu vereinheitlichen und dadurch die Geschäftsabwicklung zu vereinfachen.

Mit der in der Integrationsära zunehmenden Verbreitung von E-Business-Lösungen und der Entstehung von virtuellen Marktplätzen im Internet wird es für die in diesem Umfeld tätigen Unternehmen immer wichtiger, ihre existierenden Anwendungen und Softwaresysteme an diese neuen E-Business-Plattformen anzubinden. Ein Beispiel: Der Online-Verkaufsauftritt eines Versandhändlers im World Wide Web muss mit den internen Buchungssystemen des Unternehmens gekoppelt werden, um einen effizienten Ablauf der anfallenden Transaktionen abwickeln zu können. In der Zeit vor der allgemeinen Nutzung des Internets wurden die Bestellungen an den Händler in der Regel papierbasiert oder telefonisch vorgenommen, während der Kommunikationsära dann zum Beispiel durch den Versand einer E-Mail. Doch auch nach Erhalt der elektronischen Kauforder musste meistens immer noch jemand die Daten manuell in die Buchungssysteme des Händlers eingeben, was einen eigentlich vermeidbaren Ressourcen- und Zeitaufwand bedeutet. Besser wäre es, wenn der Kaufauftrag unmittelbar nach Empfang automatisiert verarbeitet und an die Buchungssysteme des Handelsunternehmens weitergeleitet würde.

Dieses Beispiel legt die Problemstellung offen, die ein Unternehmen lösen muss, um erfolgreich am elektronischen Geschäftsverkehr teilnehmen und damit zukunftsfähig bleiben zu können: die Integration vorhandener Anwendungssysteme untereinander und mit den E-Business-Systemen als Schnittstelle zu externen, über das Internet zu erreichenden Partnern, Zulieferern und Kunden. Dadurch könnten die Funktionalitäten der einzelnen, heterogenen Softwaresysteme gebündelt und den Geschäftspartnern über das Internet verfügbar gemacht werden. Zur Erreichung dieses Ziels sind eine Reihe von Fragestellungen zu beantworten:

1. Wie bindet man die heterogenen Systeme an? Es wird ein Verfahren benötigt, das in der Lage ist, ganz beliebige Komponenten anzusprechen und deren Funktionen aufzurufen. Dazu braucht man Schnittstellen, um den Komponenten Eingabedaten zu schicken und Ausgabedaten von ihnen entgegenzunehmen.
2. Wie kann man Geschäftsprozesse, die zwischen dem Unternehmen und seinen Kunden, Partnern und Zulieferern ablaufen, in das Integrationskonzept einfließen lassen? Die Ausrichtung von Unternehmensaktivitäten an betriebswirtschaftlichen Geschäftsabläufen soll die Effizienz und Produktivität des Unternehmens erhöhen. Diese Geschäftsprozesse durch Informationssysteme unterstützen zu lassen, verspricht weitere Produktivitätssteigerungen, bedarf aber auch einer Kombination von Workflow-Management und Integration der betrieblichen Softwaresysteme.

3. Wie überwindet man beim Kontroll- und Datenfluss Systemgrenzen ohne vermeidbaren Ressourcenverbrauch? Sollen die heterogenen Systeme gekoppelt werden, müssen sie Daten miteinander austauschen können; so verarbeitet vielleicht der Dienst des einen Systems die Resultate des anderen Systems. Ohne Integration würden dafür zusätzliche Ressourcen und Zeit benötigt, weil zum Beispiel das Personal die Daten manuell vom einen System in das andere übertragen müsste. Besser wäre es, wenn solche Kontroll- und Datenflüsse möglichst automatisiert abliefen und manuelle Interventionen der Benutzer vermieden werden könnten.
4. Wie bietet man den Kunden und Partnern flexible Zugangsmöglichkeiten zu den eigenen Systemen an? Sind die eigenen Systeme und Prozesse nach der Integration über das Internet aufrufbar, sollte dies mit den verschiedensten Endgeräten und Protokollen möglich sein, um eine möglichst große Zahl von Anwendern zu erreichen. Neben den üblichen Web-Browsern ist in Zukunft mit einem großen Bedarf an Unterstützung mobiler Geräte zu rechnen. Außerdem sind neben menschlichen Benutzern auch die Informationssysteme anderer Institutionen als mögliche Empfänger mit ihren speziellen Formaten und Protokollen in Betracht zu ziehen. Die Frage ist also, welche Möglichkeiten es gibt, die Forderung nach einem solchen Multi-Channel-Ansatz umzusetzen.
5. Wie können Entwickler und Systemingenieure des Unternehmens bei den Aufgaben der Softwareintegration unterstützt werden? Diese Frage ist zu betrachten, weil mit wachsender Größe und Anzahl der vorhandenen Systeme die zu lösenden Integrationsaufgaben schnell sehr komplex und schwierig beherrschbar werden können.

Die vorliegende Diplomarbeit behandelt vorgenannte Fragestellungen und fokussiert dabei einen prozessorientierten Ansatz, der sowohl die Anforderungen der Softwareintegration als auch der Abbildung der Geschäftsprozesse berücksichtigt. Das prozessorientierte Vorgehen bei der Integration der vorhandenen Softwarekomponenten ermöglicht nämlich eine einfache Einbeziehung der im Unternehmen ablaufenden Vorgänge. Untersucht werden soll, wie sich im Rahmen eines Workflow-Management-Systems die existierenden Systemfunktionen als Aktivitäten der Workflow-Modelle nutzen lassen. Die Abwicklung der betrieblichen Geschäftsprozesse würde dadurch effizienter, schneller, kostengünstiger und auch sicherer, weil softwaregestützte Konsistenzprüfungen in die Vorgangsbearbeitung eingebaut werden könnten. Außerdem geht die Arbeit der Frage nach, wie die Prozesse möglichst intuitiv modelliert werden könnten, so dass auch Mitarbeiter die Modelle verstehen und anlegen können, die keine technische Spezialausbildung mitbringen. Insgesamt kann die Automatisierung bislang manueller Tätigkeiten die Produktivität und damit die Konkurrenzfähigkeit des Unternehmens erhöhen.

Des Weiteren wird ein Schwerpunkt der Untersuchung auf den Vorteilen und Möglichkeiten von XML als grundlegendes Datenformat liegen. Wir betrachten, wie die Verarbeitung von XML-Dokumenten in die Workflow-Modelle eingebettet werden kann und wie sich mit Hilfe von XML der geforderte Multi-Channel-Ansatz realisieren lässt, der den Unternehmen die unterschiedlichsten elektronischen Kommunikations- und Vertriebswege zu ihren Kunden und Partnern eröffnet.

Im praktischen Teil der Diplomarbeit sollen die entwickelten Konzepte in Softwarewerkzeuge umgesetzt werden, die den Anwender umfassend bei der Erstellung, Wartung und Ausführung der Workflow-Modelle zur Softwareintegration unterstützen.

Ziel ist ein ganzheitliches System, das eine prozessorientierte Softwareintegration angefangen von der Modellierung bis hin zur Ausführung der Workflows ermöglicht. Zur praktischen Anwendbarkeit der Werkzeuge sollen sie in die E-Business-Plattform *sunShine* eingebettet werden, die Basisfunktionalitäten – zum Beispiel zum Aufbau eines Internetportals – zur Verfügung stellt. sunShine ist ein Produkt der Paderborner S&N AG, mit der die Diplomarbeit in intensiver Zusammenarbeit praxisnah und als Gruppenarbeit erstellt wurde. Die S&N AG ist vor allem als IT-Lösungsanbieter im Bereich Finanzdienstleistungen tätig und legt in jüngster Zeit einen neuen Schwerpunkt auf die Entwicklung von E-Business-Systemen unter Einsatz von XML- und Open-Source-Technologien. Aus den Erfahrungen des Unternehmens ließen sich Anforderungen ableiten, deren Erfüllung zur praktischen Einsetzbarkeit der in dieser Arbeit entwickelten Konzepte und Systeme wesentlich beigetragen hat.

Die vorliegende Arbeit gliedert sich in drei Teile: Der erste Teil umfasst die Kapitel 2 bis 5 und behandelt zunächst Workflow-Management und Softwareintegration im Allgemeinen. Nach der Beschäftigung mit den Grundlagen werden in einer Fallstudie die besonderen Anwendungsmöglichkeiten im E-Business sichtbar. Die sich daraus ergebenen Anforderungen dienen als Grundlage für die sich anschließende Evaluierung ausgewählter, vorhandener Systeme. Besonderes Augenmerk liegt dabei auf der Möglichkeit, Workflows durch ausführbare, grafische Modelle zu definieren, anstatt sie zu programmieren. In diesem Kontext werden bekannte visuelle Modellierungssprachen wie Petri-Netze, Ereignisgesteuerte Prozessketten und UML-Aktivitätendiagramme auf ihre Brauchbarkeit für diese Anwendung untersucht.

Im zweiten Teil, der mit den Kapiteln 6 bis 8 den wichtigsten Teil der Arbeit darstellt, werden die selbst entwickelten Konzepte zur Lösung der Probleme und Erfüllung der Anforderungen ausgearbeitet und vorgestellt. Um die im ersten Teil festgestellten Lücken bei den vorhandenen Ansätzen zu schließen, wird eine Erweiterung für UML-Aktivitätendiagramme definiert. Diese sogenannten Prozessdiagramme erlauben die visuelle Modellierung ausführbarer Workflow-Modelle. Da den Workflows eine implizite Verarbeitung von XML-Dokumenten zu Grunde liegt, werden Typkonzepte für XML-Dokumente entwickelt. Diese dienen als Grundlage für typisierte Schnittstellen zu den verwendeten Softwarekomponenten und typisierte Transitionen als Übergänge zwischen den einzelnen Aktivitäten. Dadurch lässt sich ein sehr hohes Maß an Wiederverwendbarkeit und Typkonsistenz für den gesamten Workflow sicherstellen.

Der dritte Teil schließlich enthält in den Kapiteln 9 bis 11??? eine Beschreibung der praktischen Umsetzung der zuvor eingeführten Lösungsansätze. Ein im Rahmen der Diplomarbeit entwickelter Editor hilft bei der Erstellung und Konsistenzprüfung der Modelle, ein Prozessinterpreter kann die Modelle ausführen. Durch die Implementierung dieser Werkzeuge und ihre Einbettung in die vorhandene E-Business-Plattform wird die praktische Anwendbarkeit der erarbeiteten Konzepte verdeutlicht.

2 Grundlagen des Workflow-Managements

Zu Beginn der Arbeit gehen wir in diesem Kapitel nach einer Einführung in die Integrationsproblematik im E-Business auf die Grundlagen des Workflow-Managements ein. Es werden einige begriffliche Grundlagen aus dem Bereich der Geschäftsprozess- und Workflow-Modellierung erläutert und eine Klassifizierung der in einem Unternehmen möglichen Prozesse vorgenommen. Besondere Beachtung finden die Aspekte der Workflow-Modellierung und der Unterstützung des Workflow-Managements durch Softwarewerkzeuge.

2.1 Prozessorientierte Integration im E-Business

Die Integration vorhandener Softwaresysteme ist für viele Unternehmen eine zunehmend wichtige Herausforderung, um ihre Konkurrenz- und Wettbewerbsfähigkeit zu erhalten. Derzeit sind die einzelnen Geschäftsanwendungen in vielen Firmen noch voneinander isoliert. Zur Verknüpfung dieser „Insellösungen“ innerhalb der betrieblichen Geschäftsprozesse sind meistens manuelle Schritte nötig, was vermeidbare, zusätzliche Finanz- und Zeitressourcen beansprucht (vgl. [Moh01] S.4).

Durch den elektronischen Handel und die dafür installierten E-Business-Systeme sollen Firmenangebote für Kunden und Handelspartner über das Internet bereitgestellt werden. In diesem Bereich gewinnt die Kopplung von betriebswirtschaftlichen Geschäftsprozessen mit informationstechnischen Netzwerken wie dem Internet oder unternehmenseigenen Intranets zunehmend an Bedeutung. Um konkurrenzfähig zu bleiben, müssen Unternehmen ihre Geschäftsvorfälle durch eine flexible E-Business Infrastruktur unterstützen. Durch eine solche webbasierte Infrastruktur können Geschäftsprozesse beispielsweise von einem Mitarbeiter an einem Internetportal des Unternehmens ausgelöst werden. Der Vorgang wird dann durch das IT-System des Unternehmens an die Mitarbeiter oder Softwaresysteme weitergeleitet, die die gewünschte Aufgabe erledigen können. Ein derart automatisierter Ablauf führt zu einer geringeren Durchlaufzeit und großer Effizienz.

Die Beobachtung der E-Business- und E-Commerce-Wirtschaft zeigt immer deutlicher, dass es nicht ausreicht, neue Marktplätze und elektronische Vertriebswege einzurichten. Um sich erfolgreich am Markt behaupten zu können, ist für die Unternehmen darüber hinaus eine Integration ihrer existierenden Anwendungen sehr wichtig. Die neuen netzbasierten Prozesse müssen mit den bestehenden Softwaresystemen kommunizieren, auf die vorhandenen Datenbanken zugreifen und die bereits realisierten Funktionen aufrufen können. Ein besonderes Augenmerk erhält zunehmend die unternehmensübergreifende Integration von Geschäftsprozessen, etwa zur Kopplung und Integration der an einer Lieferkette beteiligten Unternehmen oder Unternehmensteile (Supply-Chain-Management).

„Die Zukunft des E-Business liegt ganz klar in der Integration“, so zitiert die Computer Zeitung (siehe [Hen01]) die Beraterin eines führenden Marktforschungsunternehmens und berichtet über eine Studie des Fraunhofer-Instituts für Fabrikbetrieb

und –automatisierung, nach der „lediglich 18 Prozent aller Internet-Anwendungen […] bislang in unternehmensweite Anwendungen integriert“ sind. Hier bietet sich daher noch ein großes Optimierungspotenzial, was sich auch in den Planungen der befragten Unternehmen niederschlägt. „Die Fraunhofer-Umfrage zeigt, dass 38 Prozent der befragten Unternehmen gerade planen, ihre Einkaufsmärkte zu integrieren, 18 Prozent wollen das mit ihren Web-Shops tun. 30 Prozent der Unternehmen sind bereit, ihren Kunden und Lieferanten einen Internet-basierten Zugang zu Informationen aus ihrem ERP-System zu bieten.“ (aus [Hen01]). Damit wird klar, „dass in der zweiten Phase der E-Business-Welle, die gerade beginnt, der Schwerpunkt auf Web-Enabling und Integration liegt.“ (ebd.).

Die Thematik der unternehmensweiten Integration der verschiedensten alten (legacy) und neuen (E-Business) Anwendungen wird unter dem Schlagwort *Enterprise Application Integration* (EAI) gehandelt. „Den meisten EAI-Ansätzen gemein sind die drei Ebenen Anwendungsintegration, Transformation und Geschäftsprozessabbildung, die in dieser Reihenfolge eine Abstufung des erzielbaren Integrationsgrads im Unternehmen widerspiegeln.“ (aus [CZ01]). Bei der Anwendungsintegration wird über einheitliche Schnittstellen die Kommunikation zwischen den Anwendungen vereinfacht. Da dies aber nicht immer möglich ist, werden Transformationen benötigt, damit jede Einzelanwendung in den geforderten Formaten der anderen Systeme kommunizieren kann. Die dritte Stufe der Geschäftsprozessabbildung, auf die in dieser Arbeit besonders eingegangen werden soll, baut auf den Grundlagen der ersten beiden Stufen auf, berücksichtigt bei der Integration aber die betrieblichen Abläufe und verbindet die Einzelanwendungen zu Akteuren bei der automatisierten Durchführung von Workflows (vgl. [CZ01]).

Diese Abgrenzung der Integrationsstufen schlägt sich auch auf die jeweils bei der Umsetzung verwendeten Technologien und Architekturen nieder. In [Joh00] (S.3f) wird zwischen drei wesentlichen Vorgehensweisen unterschieden: Die einfachste Form der Applikationsintegration ist demnach eine *Punkt-zu-Punkt-Lösung*, bei der jede Anwendung unmittelbar mit jeder anderen gekoppelt wird (siehe Abbildung 2.1a). Diese Methode eignet sich allerdings nur für eine kleine, überschaubare Anzahl zu integrierender Systeme, weil die Menge der Verbindungen und Schnittstellen sonst schnell unübersichtlich und unkontrollierbar wird. Abhilfe schafft hier die Benutzung eines *Message-Brokers* (siehe Abbildung 2.1b), der als zentrale Instanz Nachrichten zwischen den einzelnen Anwendungen übermittelt und so jede Anwendung nur noch eine Schnittstelle zu dieser Instanz haben muss. Der Message-Broker muss eine eingehende Nachricht entgegennehmen, in ein für den Empfänger verständliches Format transformieren und dann an den Empfänger weiterleiten. Dieser Ansatz unterstützt damit die oben genannte zweite Integrationsstufe der Transformation.

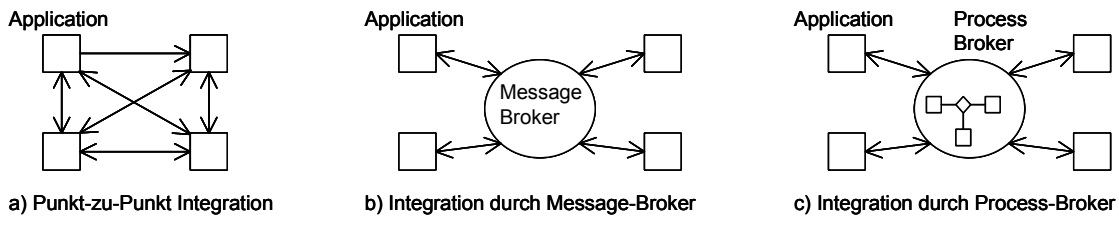


Abb. 2.1: Architekturen für EAI (aus [Joh00] S.4)

Als Erweiterung des Message-Brokers sieht [Joh00] den *Process-Broker* (siehe Abbildung 2.1c), der zusätzlich zur Formatkonvertierung auch Prozesslogik zur Verknüpfung der Anwendungen kapselt. Damit wird ein Architekturkonzept für die dritte Stufe des EAI, der Geschäftprozessabbildung, vorgeschlagen, mit der es möglich wird, die Prozesse mit einer grafischen Benutzungsschnittstelle zu analysieren und zu administrieren. „Diese Visualisierung reduziert die Komplexität und ermöglicht verschiedenen Benutzergruppen die Teilnahme am Entwurf der Prozesse. Die Process-Broker-Technologie kann als eine Weiterführung von Workflow-Management-Systemen betrachtet werden“ (übersetzt nach [Joh00] S.4).

Wir wollen bei solchen EAI-Ansätzen, die die Geschäftsprozesse des Unternehmens berücksichtigen, von *prozessorientierter Integration* sprechen. Es liegt auf der Hand, dass die umfangreichen existierenden Anwendungen eines Unternehmens zur Realisierung der neuen E-Business-Geschäftsprozesse nicht komplett neu geschrieben werden, sondern als erprobte und gut getestete Komponenten wiederverwendet werden sollen (vgl. [Moh01] S.4). Diese Komponenten müssen im Rahmen des prozessorientierten Integration zu einem Workflow zusammengesetzt werden. Beim Start eines solchen Prozesses löst die Eingabe der Workflow-relevanten Daten an einer Benutzungs- oder Systemschnittstelle einen Datenfluss aus. Diese Daten werden dann durch eine oder mehrere Anwendungen im Verlauf des Vorgangs entweder direkt verarbeitet, oder diese Anwendungen führen unterstützende Funktionen – quasi bewusste Seiteneffekte – aus, während die Daten durch den Prozess fließen (vgl. [Moh01] S.5).

Zu den Herausforderungen der prozessorientierten Integration gehören laut [Moh01] (S.5ff) die Überwindung inkompatibler Protokolle und Datenformate sowie insbesondere Verfahren zum Workflow-Entwurf und zur Workflow-Kontrolle. In [Wes01] heißt es dazu: „Diese Lösungen müssen in der Lage sein, Geschäftsprozesse zu modellieren und zu automatisieren.“ Dies hebt die Bedeutung des Workflow-Managements im Rahmen der prozessorientierten Integration hervor. Für die angesprochene Überwindung inkompatibler Datenformate eignet sich der derzeitige Standard XML (*eXtensible Markup Language*, siehe [XML00]), bei dem sich zusätzliche Kostenreduktionen ergeben können, weil ausreichend Werkzeuge zum Umgang mit XML-Daten – teilweise sogar frei – verfügbar sind (vgl. [Moh01] S.11). Insbesondere gibt es die Möglichkeit, XML-Daten auf andere proprietäre Formate abzubilden, die von älteren Anwendungen genutzt werden.

2.2 Prozesse in einem Unternehmen

Bei den betrieblichen Abläufen in einem Unternehmen kann man die auftretenden Prozesse in Materialprozesse, Informationsprozesse und Geschäftsprozesse klassifizieren (vgl. [Med93]).

Materialprozesse stellen materielle Transformationen in einem Unternehmen dar. Dazu gehört zum Beispiel die Erzeugung eines Endproduktes aus den Rohmaterialien. Zu einem Materialprozess gehören immer physische Aktivitäten an realen Objekten.

Informationsprozesse dagegen behandeln die Verarbeitung von Daten, Informationen und die damit verbundenen Vorgänge. Sie sind oft durch Softwaresysteme automatisiert oder werden teilautomatisiert von Sachbearbeitern mittels Rechnerunter-

stützung erledigt. In diesen Bereich fallen zum Beispiel alle Buchungsvorgänge in einem Unternehmen, die Rechnungslegung usw.

Geschäftsprozesse (engl.: business processes) sind markt- oder kundenorientierte Prozesse, die einen messbaren Nutzwert erzeugen oder einen Beitrag zur Erreichung von Unternehmenszielen bringen sollen. Der Begriff des Geschäftsprozesses ist dabei ein umfassenderer, denn Geschäftsprozesse können sowohl Material- als auch Informationsprozesse als Teilprozesse enthalten und durch sie realisiert werden. In [Wen00] (S.8) wird ein Geschäftsprozess daher folgendermaßen definiert:

> „Unter einem Geschäftsprozeß soll der potentielle zeitlich-logische Ablauf von Aktivitäten zur Transformation von Material und/oder Information verstanden werden, die zur Erfüllung eines geschäftlichen Ziels notwendig sind."

Während einige Autoren die Begriffe Geschäftsprozess und Wertschöpfungskette synonym benutzen, sehen andere Geschäftsprozesse als Teil der Wertschöpfungskette an. Ein Geschäftsprozess kann auch organisatorische Grenzen wie Abteilungen oder ganze Unternehmen überqueren. Er beschreibt alle Aktivitäten, mit denen für interne wie externe Kunden eine Leistung von Wert produziert wird. Um dieses Ergebnis zu erzeugen, benötigt man zur Durchführung des Geschäftsprozesses Ressourcen wie Material und Informationen (nach [Kel99]). Da in dieser Arbeit der Fokus auf der rechnerunterstützten Ausführung von Prozessen liegt, mit denen elektronische Geschäftsvorfälle im Rahmen des E-Business bearbeitet und dabei vorhandene Softwaresysteme eingebunden werden sollen, wollen wir die Materialprozesse nicht näher untersuchen und uns detaillierter den Informationsprozessen widmen.

2.2.1 Der Begriff „Workflow"

Geschäftsprozesse werden in einem Unternehmen im Rahmen des sogenannten *Business Process Engineering* untersucht, analysiert und optimiert. Ziel ist dabei eine prozessorientierte Ausrichtung der Geschäftsvorfälle im Unternehmen, um eine bessere Kundenorientierung, höhere Effizienz, höhere Produktqualität, reduzierte Kosten und die Wahrnehmung neuer Herausforderungen und Chancen zu erreichen (vgl. [Geo95] S.120). Diese Beschäftigung mit den unternehmenseigenen Prozessen wird zunächst unabhängig von Informationssystemen durchgeführt, die zur Automatisierung der Prozesse dienen könnten. Das Ergebnis der Betrachtung ist eine Beschreibung der Geschäftsprozesse, die meist in papierbasierter Form vorliegt, und sich mit der Frage befasst: *„Was ist zu tun?"* (vgl. [Wen00] S.9).

Um die oben genannten Ziele der Prozessanalyse zu erreichen, ist es aber auch erforderlich, sich mit der Umsetzung der Ergebnisse und der Durchführung der optimierten Prozesse zu befassen. Hier ist die Frage *„Wie ist es zu tun?"* zu beantworten. Besonders die wachsenden Möglichkeiten der Informationstechnik bieten ein großes Potenzial für die effiziente Ausführung von Prozessen. Dazu ist allerdings eine genauere Beschreibung der einzelnen Aufgaben innerhalb eines Geschäftsprozesses nötig, die auch aus Sicht der vorhandenen Informationssysteme erfolgen muss, mit denen die Aufgaben koordiniert und ausgeführt werden sollen (siehe [Wen00] S.9).

Diese Beschreibung der Aufgaben und der Abläufe, die diese Aufgaben im Sinne des zu Grunde liegenden Prozesses anordnen, kennzeichnet den Begriff des

Workflow. Nach Wenski (siehe [Wen00] S.10) soll unter Workflow Folgendes verstanden werden:

> „Ein Workflow ist eine partiell oder vollständig geordnete Menge von Tasks. Tasks wiederum bilden die Basisbauelemente für Workflows. Ein Task ist eine partiell oder vollständig geordnete Menge von Operationen, Anweisungen für von Menschen auszuführenden Aufgaben oder anderen Tasks (Hierarchisierung). Ein Task besteht auf unterster Hierarchieebene aus Aktivitäten, die von Aufgabenträgern ausgeführt werden."

Diese Definition erklärt einen Workflow als hierarchisch strukturierte, geordnete Menge von Aufgaben, die letztlich auf unterster Ebene als Aktivitäten beschrieben werden, die von einer Instanz ausgeführt werden können. Als Aufgabenträger können sowohl Menschen als auch technische Systeme und Software in Frage kommen. Leider wird in dieser Definition die Abgrenzung zur vorausgegangenen Definition eines Geschäftsprozesses nicht sehr deutlich. Bei einem Workflow geht es um die Umsetzung und Ausführung eines Geschäftsprozesses. Im Rahmen dieser Arbeit liegt ein besonderer Fokus auf der computer- und softwaregestützten Automatisierung der Prozessdurchführung. In diesem Kontext sei daher ergänzend die Workflow-Definition aus dem Workflow-Referenzmodell der Workflow-Management-Coalition (WFMC) zitiert, die Workflows bewusst im Zusammenhang mit informationstechnischer Ausführung betrachtet. In [Wmc95] (S.6) heißt es zur Definition des Workflow-Begriffs:

> „The computerized facilitation or automation of a business process, in whole or part."

Als Workflow soll im Folgenden daher eine detaillierte Beschreibung von Prozessen, den darin enthaltenen Aktivitäten sowie den Ablaufbeziehungen der Aktivitäten untereinander verstanden werden. Der Begriff Workflow wird dabei häufig als Bezeichnung sowohl für die Beschreibung des Vorgangs als auch für den Vorgang selbst benutzt. Ist eine explizite Unterscheidung nötig, kann man von *Workflow-Modell* auf der einen Seite und *Workflow-Instanz* auf der anderen Seite sprechen.

Wenn, wie in der Definition der Workflow-Management-Coalition gefordert, ein Workflow durch Informationssysteme unterstützt werden soll, indem zum Beispiel die Abfolge der einzelnen Arbeitsschritte gesteuert und überwacht sowie entsprechende Aufgabenträger für die Aktivitäten zugewiesen werden, benötigt man ein *Workflow-Management-System* (WFMS). Ein solches System wird im Referenzmodell (siehe [Wmc95] S.6) definiert als:

> „A system that completely defines, manages and executes "workflows" through the execution of software whose order of execution is driven by a computer representation of the workflow logic."

Im später folgenden Abschnitt über die Unterstützung durch Software-Werkzeuge werden wir auf die Anforderungen und Eigenschaften eines solchen Systems noch näher eingehen.

2.2.2 Klassifizierung von Informationsprozessen

Je nach dem, wie gut sich Informationsprozesse durch Rechner unterstützen lassen, wie hoch der Grad ihrer Wiederholbarkeit liegt und wie stark sie sich kontrollieren lassen, spricht man von unstrukturierten oder gut strukturierten Prozessen. Diese Klassifizierung korrespondiert mit der Einteilung von Informationsprozessen in eher *„Mensch-orientierte“* auf der einen Seite und *systemorientierte* Prozesse auf der anderen Seite. Diese Prozesstypen bilden bei der in [Geo95] vorgestellten Charakterisierung die Extrema auf einer kontinuierlichen Skala mit fließenden Übergängen.

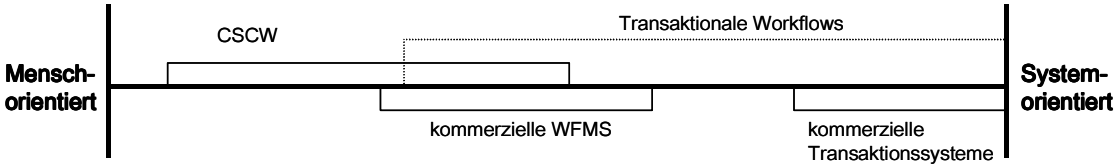


Abb. 2.2: Kontinuum von Mensch-orientierten zu systemorientierten Workflows (aus [Geo95])

Bei den Mensch-orientierten Informationsprozessen geht es um die Unterstützung von Mitarbeitern, die an einem Prozess beteiligt sind, gemeinsam eine Aufgabe zu lösen haben und ihre Aktivitäten koordinieren müssen. Da hierbei Menschen, und nicht Softwaresysteme, in überwältigender Mehrheit Aufgabenträger sind, zeichnen sich die Prozesse durch einen hohen Grad an *Interaktivität* der Benutzer mit dem WFMS aus. Eine Modellierung solcher Prozesse und die anschließende rechnerbasierte Unterstützung bei der Ausführung soll die Zusammenarbeit der betroffenen Personen und ihre Interaktion mit dem Computer erleichtern. Eine wesentliche Aufgabe des Prozessmodells ist es daher, eine Zuordnung der Teilaufgaben an einzelne Aufgabenträger vorzunehmen. Dabei müssen sowohl die jeweiligen menschlichen Fähigkeiten als auch die Anforderungen der Aktivitäten berücksichtigt werden. Auch die Gegebenheiten in den Büros und an den Arbeitsplätzen sowie die jeweils übliche Art der Zusammenarbeit haben Auswirkungen auf die Ausprägung der Mensch-orientierten Prozessmodelle (vgl. [Wen00] S.6f). Solche Fragestellungen der teamorientierten Aufgabenabwicklung sind Gegenstand des Forschungsgebietes des *Computer Supported Cooperative Work (CSCW)*. Dort wird die spezielle Klasse von Computersystemen untersucht, die Gruppenarbeit und andere Formen der menschlichen Zusammenarbeit durch Koordination, Kooperation und Kommunikation unterstützen. Daher befasst sich CSCW auch mit dem Management von Mensch-orientierten, interaktiven Workflows.

Mensch-orientierte Workflow-Modelle haben Informationen über die Prozesssemantik, d.h. sie wissen zum Beispiel, in welcher Abfolge ein Dokument weitergeleitet werden muss. Das Prozessmodell impliziert aber kein genaues Wissen über die Dateninhalte des Dokuments und die Semantik der zu verarbeitenden Daten. Darum kann ein Workflow-Management-System hier nicht die inhaltliche Konsistenz der Daten sicherstellen. Diese Aufgabe muss von den beteiligten Menschen wahrgenommen werden.

Bei den systemorientierten Prozessen dagegen kann man in das Modell des Workflows Informationen über die Semantik und die Struktur der zu verarbeitenden Daten integrieren. Dadurch gewinnt das WFMS bei der Ausführung des Modells an

Verantwortung für die Aufrechterhaltung der Konsistenz von Daten und Prozessablauf. Die meist hochautomatisierten Prozesse enthalten statt Aufgaben, die einem Mitarbeiter zugeordnet werden, Aktivitäten, die von Softwarekomponenten ausgeführt werden. Die ausführende Instanz, also das WFMS, kontrolliert die Software-Aktionen bei gar keiner oder nur gelegentlicher Benutzer-Intervention und muss die Abarbeitung des Workflows gemäß der zugehörigen Workflow-Modelle und -Definitionen gewährleisten (vgl. [Geo95] S.128).

Transaktionale Workflows

Wird gefordert, dass die Workflows Transaktionseigenschaften wie Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (ACID-Eigenschaften, siehe [Bac98] S. 440) erfüllen sollen, spricht man von *transaktionalen Workflows*. Die bekannten ACID-Eigenschaften können in der Regel nur bei systemorientierten Workflows eingefordert werden. Bei Geschäftsprozessen auf höherer Ebene, die oft viel langlebiger sind, kann es zu einem Spannungsfeld zwischen den Transaktionseigenschaften und den Anforderungen an das Workflow-Management kommen. So ist hier die Forderung nach Atomarität problematisch, denn es wäre beispielsweise nicht sinnvoll, den monate- oder jahrelangen Prozess der Entwicklung einer neuen Maschine auf den konsistenten Startzustand zurückzusetzen, wenn ein Teil des Systems abgestürzt ist (vgl. [Wen00] S.17f). Im Bereich technischer, systemorientierter Workflows macht eine Anwendung des Transaktionskonzepts dagegen möglicherweise Sinn. In [Böh95] wird allerdings in diesem Zusammenhang festgestellt: „Die in Datenbankanwendungen seit Jahrzehnten etablierten Techniken haben bei WFMS noch bei weitem nicht die Verbreitung, wie es notwendig wäre, um bestimmte Eigenschaften bei der Ausführung von Vorgängen zuzusichern."

Die Zusicherung von Atomarität bedarf einer Möglichkeit, gescheiterte Vorgänge zurückzusetzen und ihre Wirkung rückgängig zu machen. Diese Aufgabe gestaltet sich für ein WFMS sehr schwierig, besonders wenn für die Erfüllung von Teilaufgaben vorhandene Softwaresysteme eingesetzt werden. Für das Rücksetzen eines transaktionalen Workflows wäre es nötig, dass auch alle angesprochenen Softwarekomponenten eine Undo-Operation unterstützen und ihre jeweilige Teilaufgabe rückgängig machen können (vgl. [Geo95] S.138). Noch 1995 kommt man in [Geo95] zu der Feststellung: „No commercial WFMSs we are aware of offers significant support for workflow recovery." Dies mag sich inzwischen geändert haben; es macht jedoch deutlich, wie schwierig die Gewährleistung von Transaktionseigenschaften für Workflows ist.

2.2.3 Integration existierender Anwendungen

Das Modell eines systemorientierten Prozesses muss die einzelnen Prozessaktivitäten zwecks automatisierter Ausführung existierenden Applikationen (sog. *legacy applications*) oder Softwarekomponenten zuordnen. Damit wird die Interoperabilität der vorhandenen, meist heterogenen Softwaresysteme untereinander, aber vor allem mit dem WFMS, zu einer zentralen Frage bei der Prozessmodellierung und –ausführung. Das WFMS muss auf die heterogenen, autonomen und oft verteilten Komponenten zugreifen sowie den Kontroll- und Datenfluss zwischen diesen Einheiten steuern können. Im Einzelnen sind drei Formen der Einbindung vorstellbar (vgl. [Böh95] S.15), die auch

beliebig miteinander kombiniert werden können:

- **Einbindung zur Ausführung einer Funktion oder Aktivität:** Hierbei geht es um den automatischen Aufruf einer externen Komponente zur Durchführung eines Bearbeitungsschritts. Denkbar sind hier die Benutzung elementarer Einheiten wie zum Beispiel JavaBeans oder auch der Aufruf größerer Module, zum Beispiel aus einer betriebswirtschaftlichen Standardsoftware. Neben einer Möglichkeit, die jeweilige Komponente aufzurufen (z.B. über CORBA, RMI, RFC, Internet-URL) müssen die beiden folgenden Punkte der Übergabe von Daten und der Rückgabe von Ergebnissen gelöst werden.
- **Übergabe von Parameterwerten und Ergebnissen an existierende Applikationen und Datenbanken:** Beim Aufruf von Funktionen bestehender Software müssen in der Regel Parameterwerte übergeben werden. Außerdem ist es oft nötig, Zwischenergebnisse, Protokollinformationen oder sonstige Datensätze an externe Anwendungen oder Datenbanken zwecks Weiterverarbeitung oder Speicherung zu übergeben. Zu diesem Zweck müssen die Komponenten, die Ziel dieses Datentransfers sind, eine Schnittstelle zum Empfang der Daten bereitstellen. Gegebenenfalls ist außerdem noch eine Konvertierung des Datenformats nötig, wenn ein spezielles, zum internen Format des Workflow-Management-Systems inkompatibles Datenformat von der Zielkomponente gefordert wird.
- **Übernahme von Daten aus externen Quellen wie Datenbanken und Applikationen:** Ein analoges Problem zur Übergabe von Daten beim Benutzen der existierenden Komponenten ist das Entgegennehmen der Ergebnisse, nachdem eine Komponente eine gewünschte Funktion ausgeführt oder eine Datenbank eine bestimmte Anfrage durchgeführt hat. Schwierig wird diese Aufgabe bei einer großen Anzahl verschiedener Datenbanken, wie sie sich in großen, verteilten Unternehmen finden lassen. Spezielle Treiber und Konvertierungsprogramme müssen unter Umständen für die Ansteuerung der Anwendungen und Datenbanken sowie für die Konvertierung und Aufbereitung der Daten sorgen.

Nach Böhm (siehe [Böh95]) stellt die Integration bestehender Anwendungen „ein nicht zu unterschätzendes Problem bei der Einführung von Workflow-Management-Systemen“ dar, weil für den Daten- und Informationsaustausch häufig Schnittstellenerweiterungen zu den existierenden Anwendungen mit großem Aufwand realisiert werden müssen. Mit dieser Problematik werden wir uns auch in dieser Arbeit beschäftigen, denn hier soll es gerade ein Hauptzweck der Workflow-Modelle sein, die Integration der existierenden Softwarekomponenten zu fördern.

2.3 Modellierung von Prozessen

Soll die Ausführung von Informationsprozessen durch Software unterstützt oder ganz vorgenommen werden, muss zunächst eine Modellierungs- und Analysephase vorausgehen (siehe [Wen00] S.8). In der Modellierungsphase werden die Prozesse in einer semi-formalen Darstellung zu einem Modell abstrahiert. Ein statisches Organisationsmodell hilft bei der Findung und Zuweisung geeigneter Aufgabenträger. Bei der Analyse werden die Prozessmodelle auf mögliche kritische Pfade, Engpässe, Durchlaufzeiten und Inkonsistenzen untersucht.

2.3.1 Organisationsmodellierung und statische Modelle

Im Zusammenhang mit der Modellierung ist neben der Definition der Workflow-Modelle auch eine *Organisationsmodellierung* durchzuführen, mit der „*strukturelle*, *funktionelle* und *technische* Aspekte“ (siehe [Böh95] S.4) der Organisation festgehalten werden, die die Prozesse ausführen soll. Die Organisationsstruktur muss beschrieben werden, damit das WFMS eine Zuordnung der einzelnen Aktivitäten eines Workflows an einen Bearbeiter vornehmen kann. Für Mensch-orientierte Prozesse werden in den strukturellen Modellen vor allem Akteure, Organisationseinheiten, Gruppen, Stellen und Rollen spezifiziert und miteinander verknüpft. Das Konzept der Stellen und Rollen ist dabei nützlich, um die Definition der einzelnen Workflow-Modelle zu anonymisieren, also unabhängig von konkreten Mitarbeitern zu gestalten. Vielmehr werden Aufgaben an abstrakte Stellen delegiert, denen mit Hilfe des Organisationsmodells konkrete Personen zugeordnet sind. In [Böh95] werden außerdem verschiedene Arten von Beziehungen zur Beschreibung der Organisationselemente in der statischen Struktur ergänzt. Dazu gehören Aspekte der Unternehmenshierarchie, Zugehörigkeit zu Arbeitsgruppen, Bestimmung von Vorgesetzten und Vertretern sowie von Kompetenzen und Zuständigkeiten.

Die Forderung nach einem ausführlichen Unternehmens- oder Organisationsmodell ist vor allem zur Realisierung betriebswirtschaftlicher Geschäftsprozesse von großer Bedeutung, bei denen eine automatische Zuordnung von Aufgaben an dafür zuständige Stellen und geeignete Bearbeiter gewährleistet werden muss. Bei system-orientierten Prozessen, wie sie in dieser Arbeit betrachtet werden sollen, schwindet dagegen die Bedeutung der Mitarbeiter als Aufgabenträger zugunsten von Softwarekomponenten, die mit der Erfüllung der einzelnen Aktivitäten beauftragt werden.

Trotzdem ist für das Workflow-Management-System ein strukturelles bzw. statisches Modell der Aufgabenträger nötig, auch wenn es sich bei diesen Aufgabenträgern um Softwarekomponenten handelt. Nur solche Komponenten, die im Strukturmodell aufgeführt sind, können bei der Definition des Workflows für die Übernahme einer Aktivität eingesetzt werden. Dabei muss in dem statischen Modell spezifiziert sein, wie die Komponente bezeichnet wird, wie sie angesprochen wird, welche Funktionen sie zur Verfügung stellt und welche Parameter sie erwartet.

2.3.2 Vorgangsmodellierung und dynamische Modelle

Nach der Spezifikation des Strukturmodells können die dort aufgeführten Aufgabenträger bei der Definition der Workflow-Modelle benutzt werden. Ein Workflow-Modell, in der Literatur auch als *Vorgangstyp* bezeichnet, beschreibt nach [Böh95] zum einen die Ablaufreihenfolge der Aufgaben und zum anderen die einzelnen Aktivitätstypen. Zu den wichtigsten Kontrollstrukturen für die Beschreibung der Abläufe zählen laut [Böh95] die Konstrukte Sequenz, Parallelität und Verzweigung (vgl. Abschnitt 3.3 über die Anforderungen an eine visuelle Prozessmodellierungssprache).

Zur Modellierung der Aktivitäten innerhalb der Abläufe ist die Angabe der zugehörigen Aufgabenbeschreibung und der zuständigen Bearbeiter nötig. Aktivitäten können mit einer *losen Kopplung* an Anwendungsprogramme gebunden werden, wenn ein Sachbearbeiter das Programm zur Erfüllung der Aufgabe aufrufen und benutzen kann. Bei einer *engen Kopplung* von Aktivität und Programm wird die Applikation automatisch gestartet. Sie übernimmt an Stelle eines Sachbearbeiters die Erfüllung der Aufgabe, etwa die Durchführung einer komplexen Berechnung auf einem Großrechner (vgl. [Böh95], S.8).

Um verschiedene Abstraktionsebenen einführen zu können, ist bei den meisten Modellierungsansätzen eine Möglichkeit zur *Schachtelung von Aufgaben* enthalten. So kann eine Aufgabe in einer Hierarchisierung in immer feinere Teilaufgaben zerlegt werden, bis diese schließlich als atomare Dienste vorhandener Systeme oder menschlicher Aufgabenträger in Anspruch genommen werden können. Die Modelle auf einer höheren Abstraktionsstufe fassen mehrere Teilaufgaben zu einem Teilprozess zusammen und erlauben so einen besseren Überblick, besonders bei umfangreichen Workflow-Modellen (siehe [Geo95], S.135).

Ein Vorgangs- oder Workflow-Modell ist die Grundlage für die nachfolgende Analysephase, in der logische Fehler und Engpässe bei der Prozessdurchführung gefunden und beseitigt werden sollen. Hilfreich ist hierbei die Erweiterung des dynamischen Modells um eine zeitliche Komponente, um für einzelne Aufgaben oder Prozesse zeitliche Höchstdauern, sowie Start- oder Endzeitpunkte festzulegen. Nach der Analysephase dient das Modell und die darin enthaltene Prozessdefinition dem Workflow-Management-System als Ausgangspunkt für die Durchführung des Workflows.

2.3.3 Analyse und Simulation der Modelle

Im Rahmen der Modellanalyse unterscheidet man zwischen *Testen*, *Analysieren* und *Überwachen* von Workflows (siehe [Geo95], S.136). Durch die Eingabe von Beispiel-Daten kann man die Ausführung eines Workflows simulieren, um logische Fehler zu finden und die erwartete Ausführungsdauer abzuschätzen. Übliche Testverfahren aus der Software-Entwicklung können auf Workflow-Modelle übertragen werden: Entweder gibt es eine formale Spezifikation zu einem Workflow oder man testet gegen das vom Modellierer intendierte Verhalten. Verschiedene Testfälle sollen möglichst alle Pfade des Workflows mindestens einmal abdecken. Auch Ausnahmezustände und Sonderfälle sollen dabei untersucht und bewusst erzwungen werden, um das Verhalten des WFMS in diesen Situationen zu simulieren.

Bei der Analyse von Workflow-Modellen ist das Ziel, mögliche Engpässe bei der Ausführung aufzudecken. Neben der Untersuchung der Modell-Spezifikation können Statistiken über bisherige Ausführungen und Simulationen herangezogen werden. Diese Statistiken enthalten zum Beispiel Angaben über die Ausführungsdauer oder den Durchsatz an Daten durch eine bestimmte Aktivität des Workflows. Als Ergebnis der Analyse können Verbesserungsvorschläge gemacht werden, an welchen Stellen das Workflow-Modell optimiert werden könnte.

Nach Simulation und positiver Analyse eines Modells kann es vom WFMS ausgeführt werden. Bei der Ausführung muss das Fortschreiten des Prozesses überwacht werden. Dadurch können temporär entstehende Engpässe entdeckt und durch entsprechende Gegenmaßnahmen aufgelöst werden. Während der Überwachung werden dazu Informationen benötigt, welche Workflow-Instanzen zur Zeit aktiv sind, in welchem Zustand sie sich befinden und welcher Aufgabenträger gerade welche Aktivität durchführt. Als Nebeneffekt der Überwachung können Statistiken über die Abläufe aufgestellt werden, die bei der nachträglichen Untersuchung von Fehlern hilfreich sein und außerdem als Datenquelle für die nächste Analysephase dienen können.

2.4 Unterstützung durch Software

Für die Durchführung der Modellierung und Analyse gibt es eine Reihe von Werkzeugen. Sie reicht von reinen Zeichenprogrammen, mit denen sich lediglich die Prozessmodelle durch Flussdiagramme oder ähnliche Techniken visualisieren lassen, bis zu hochintegrierten Werkzeugen wie zum Beispiel das ARIS-Toolset, mit denen sich gesamte Organisationen und Informationssysteme sowohl aus statischer wie dynamischer Sicht modellieren lassen (vgl. [Wen00] S.9).

Die meisten dieser Werkzeuge zur Modellierung der Geschäftsprozesse bewegen sich auf einer relativ abstrakten Ebene, was sich durch den Gebrauch teilweise informeller Notationen zur Erstellung der Modelle auszeichnet. Mit zunehmender Abstraktion kann die Beschreibung unpräziser und in ihrer Interpretation mehrdeutig werden. Das Auslassen von zunächst weniger wichtigen Details erleichtert zwar das Verständnis der Prozessmodelle, macht ihre Beschreibung aber unvollständig im Sinne einer Eingabe für einen ausführenden Interpreter. Diesen Unterschied zwischen dem abstrakten Modell und einer präzisen, ausführbaren Formalisierung bezeichnet man als *semantische Lücke* (s. [Wen00]). Erzeugt wird diese Diskrepanz insbesondere dadurch, dass die Semantik der zumeist grafischen Notation, in der die Prozessmodelle aufgestellt werden, nicht präzise und eindeutig definiert worden ist (vgl. z.B. Ereignisgesteuerte Prozessketten, Abschnitt 8.2). Einige Modellierungsansätze erlauben sogar die natürlichsprachliche Beschreibung auszuführender Aktivitäten, was die semantische Lücke weiter vergrößert, denn für eine softwaregestützte Ausführung müssen diese Aktivitäten genau und in einer vom Rechner verständlichen Form spezifiziert sein. Doch auch wenn die Werkzeuge nur die abstrakte Modellierung der Prozesse leisten und wegen der semantischen Lücke keine Ausführung der Modelle anbieten, kommt ihnen eine wichtige Bedeutung zu, da die von ihnen geleistete Visualisierung der Abläufe zumindest Schwachstellen erkennen lässt und besseres Verständnis für die Zusammenhänge erzeugt.

Um die beschriebene semantische Lücke zwischen dem Modell eines Geschäftsprozesses und einer präzisen Workflowbeschreibung, die als Eingabe für die Aus-

führung dienen kann, zu überbrücken, ist eine Verfeinerung der Prozessmodelle nötig. Zu den Aufgaben, die beim Übergang vom Geschäftsprozess- zum Workflow-Modell auszuführen sind, zählen nach [Wen00] (S.11f)

- die genauere Spezifizierung der informellen, oft verbal beschriebenen Aktivitäten, besonders wenn es sich um automatisierbare Aktionen handelt,
- die Strukturierung der Daten in Entitäten und Attribute und die Festlegung der Datenflüsse,
- die genaue Definition der logischen Regeln zu Verzweigungen der Kontrollflüsse, auch in Abhängigkeit von den Ausprägungen der Datenstruktur,
- die Einordnung der Bearbeiter in ein Rollen- und Qualifikationskonzept zwecks eindeutiger Zuordnung der Aufgaben.

Bei der Umwandlung eines Geschäftsprozessmodells in ein Workflow-Modell kommen zwei Vorgehensweisen in Frage: Einerseits könnte man zur Modellierung der Geschäftsprozesse eine Sprache benutzen, die so detailliert und formal ist, dass die mit ihr erzeugten Modelle als Workflow-Modelle angesehen und als Eingabe für eine ausführende Einheit genutzt werden können. Dabei muss allerdings ein höheres Maß an Komplexität bei der Modellierung in Kauf genommen werden.

Andererseits könnte man die beiden Modelle getrennt voneinander in zwei verschiedenen Sprachen mit unterschiedlichem Abstraktionsgrad aufstellen. Mit einem Konvertierungswerkzeug würde man, um sich das Aufstellen des zweiten, formaleren Modells zu sparen, das abstraktere Modell weitgehend automatisch in die Workflow-Spezifikation konvertieren. Dabei kann es allerdings unter Umständen zu Konvertierungsfehlern kommen, weil das Quell- und Zielformat der Konvertierung unterschiedliche Intentionen verfolgen und durch die semantische Lücke voneinander getrennt sind.

Wie später gezeigt wird, haben wir einen Ansatz gewählt, der beide Alternativen miteinander kombiniert. Es findet zwar eine Übersetzung des visuellen Prozessmodells in eine formale, XML-basierte Prozessbeschreibung statt, damit diese Überwindung der semantischen Lücke aber nicht zu Fehlern führt, wird die verwendete Modellierungssprache zuvor mit ausreichend Details angereichert. Außerdem erleichtert der Einsatz von wohldefinierten Schnittstellen zu den im Workflow angesprochenen Softwarekomponenten sowie eine feine Granularität der Vorgänge eine automatisierte Ausführung.

Nach der Modellierungsphase kann bei der Analyse des Modells auf Werkzeuge zurückgegriffen werden, die bei den in Abschnitt 2.3.3 vorgestellten Tests und Simulationen helfen. Danach wird ein Workflow-Management-System eingesetzt, um die spezifizierten Prozesse während der Durchführung zu kontrollieren. Im Sinne des CSCW unterstützt ein WFMS damit zeitlich und räumlich verteilte Arbeitsabläufe. In Verbindung mit den Modellierungsschritten hilft es bei der Trennung von Arbeitsorganisation und –durchführung. Außerdem lässt sich mit einem WFMS der Ablauffortschritt eines Prozesses überwachen. Das WFMS interpretiert die Spezifikation des Prozesses aus dem Workflow-Modell und steuert die Ausführung, indem der Vorgang sukzessive von Aufgabe zu Aufgabe bzw. zu den jeweiligen Aufgabenträgern weitergeleitet wird. Dabei hat das WFMS Zugriff auf benötigte Datenbanken, auf die Organisationsstruktur, also das statische Modell der Unternehmung, und es kann mit Aufgabenträgern wie Mitarbeitern oder Softwarekomponenten kommunizieren, ihnen

gemäß der Prozessvorschrift Aufgaben zuweisen und auf deren Erledigung warten. Man sagt auch, das WFMS erzeuge eine *Instanz* des Workflow-Modells und führe diese aus. Mit einem WFMS sollte es möglich sein, gleichzeitig mehrere Ausprägungen bzw. Instanzen von Workflow-Modellen auszuführen.

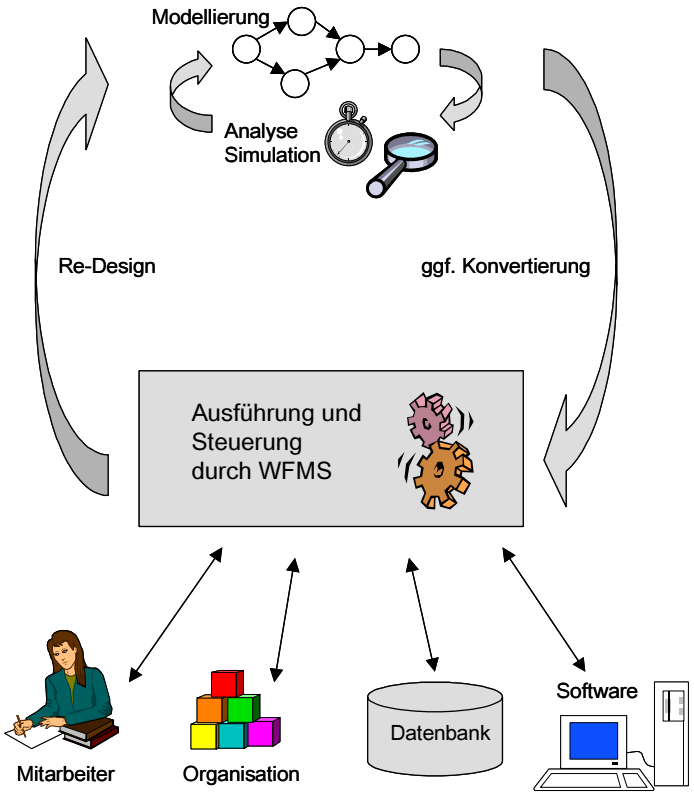


Abb. 2.3: Werkzeug-Einsatz beim Workflow-Management

Der Einsatz eines WFMS eignet sich vor allem für gut strukturierte Prozesse, die a priori spezifiziert und festgelegt werden können. Es sollte aber auch die Möglichkeit bestehen, Prozessmodelle an neue Gegebenheiten anzupassen und flexibel – eventuell bei langlebigen Prozessen sogar zur Laufzeit der Prozessdurchführung – zu ändern. Dies betrifft sowohl die Ablaufstruktur als auch die in der Prozessbeschreibung enthaltene Geschäftslogik, die den Ablauf regelt oder zumindest beeinflusst. Um dieses sogenannte *Redesign* für den Planer möglichst einfach zu gestalten, sollte es möglich sein, gewünschte Prozessänderungen mit dem bekannten Modellierungswerkzeug vornehmen zu können.

Nach der vorausgegangenen Einführung in die Thematik des Workflow-Managements sollen im Rest der Arbeit bestehende Ansätze zur prozessorientierten Integration heterogener Softwaresysteme vorgestellt und verglichen, sowie neue unter besonderer Einbeziehung von XML-Technologien entwickelt werden. Dazu werden im folgenden Kapitel zunächst anhand einer praktischen Anwendung Anforderungen herausgearbeitet, die heute an ein Workflow-Management-System in Verbindung mit einer E-Business-Plattform gestellt werden, wenn damit gleichzeitig existierende Softwarekomponenten in die Workflows integriert werden sollen.

3 Anforderungsanalyse

Mit der folgenden Anforderungsanalyse sollen die Ziele dieser Diplomarbeit abgesteckt werden. Neben allgemeinen Anforderungen für Workflow-Management-Systeme werden aus einer Fallstudie heraus spezielle Anforderungen aufgezeigt, die von besonderer Relevanz beim praktischen Einsatz solcher Systeme im E-Business sind. Da der Modellierung von Workflows eine große Bedeutung zukommt, werden die Anforderungen an eine passende Modellierungssprache gesondert betrachtet. Am Ende des Kapitels folgt schließlich eine Gewichtung der einzelnen Anforderungen und die sich daraus ergebende Zielsetzung für die später vorgestellten Konzepte.

3.1 Fallstudie: E-Business-Plattform eines großen Finanzunternehmens

In dieser Fallstudie sollen anhand eines Pilotprojektes bei der Deutschen Bausparkasse Badenia AG, einem führenden Finanzunternehmen unter den privaten Bausparkassen in Deutschland, die Ziele, Anforderungen und Architekturvorschläge vorgestellt werden, die dieses Unternehmen für den Aufbau einer E-Business-Plattform in einem Anforderungskatalog (aus [Bad01]) festgehalten hat. Da eine zentrale Komponente der neuen Plattform ein Workflow-Management-System sein soll, wollen wir auf diese Weise untersuchen, welche Anforderungen sich in der Praxis an solche Systeme ergeben.

3.1.1 Projektziele

Mit der Durchführung des Pilotprojektes strebt die Badenia Bausparkasse den Aufbau einer flexiblen und hochmodernen E-Business-Plattform an. Dabei sollen zunehmend konventionelle Informationsprozesse auf Formular- und Papierbasis durch elektronische Prozesse ersetzt werden, die über das Internet oder das firmeneigene Intranet aufgerufen werden. Statt der Arbeit mit Formularen und der anschließenden manuellen Eingabe der Daten zur Verarbeitung mit den existierenden Softwaresystemen (sog. *Back-End-Systemen*) will man in Zukunft die Daten über unterschiedlichste elektronische Schnittstellen ein- und ausgeben und mit den Back-End-Systemen kommunizieren können. Neben der Rüstung für den wachsenden Bedarf an E-Business-Unterstützung seitens der Kunden und Geschäftspartner, die die elektronischen Informations- und Vertriebskanäle nutzen möchten, verspricht sich das Finanzunternehmen von der Durchführung außerdem eine Optimierung der internen Workflows, die somit von Anfang bis Ende vollständig elektronisch abgewickelt werden können. Beide Gründe sind entscheidende Faktoren, um im zukünftigen E-Business-Zeitalter gegenüber den Konkurrenten wettbewerbsfähig zu bleiben. Die Straffung und Optimierung der Workflows sowie ihre vollständig elektronische und automatisierte Abwicklung kann zudem bereits mittelfristig zu Kosteneinsparungen führen.

Zielgruppen für die neue Plattform sollen sowohl Teile der Kunden als auch die Mitarbeiter des Unternehmens im Innen- und Außendienst sein. Als Beispiele für mögliche Anwendungen durch den Kunden werden Funktionen wie die Änderung der Daten

über den Einzug regelmäßiger Beiträge von einem anderen Konto oder die Erfassung bzw. Änderung von Freistellungsaufträgen genannt. Gerade durch Rationalisierungen im Filialsystem müssen Finanzinstitute ihren Kunden immer mehr die Möglichkeit bieten, ihre Bankgeschäfte per Internet von zu Hause aus zu erledigen. Mit wachsender Verbreitung der Internetnutzung in der Bevölkerung wächst gleichzeitig bei den Kunden die Akzeptanz dieser Kommunikationsmöglichkeit. Für Mitarbeiter des Unternehmens wird es einfacher, Vorgänge oder Änderungen in bestimmten Datensätzen über Webformulare oder andere Schnittstellen zu veranlassen.

3.1.2 Anforderungen

Vereinfacht gesehen sollen auf der geplanten E-Business-Plattform die elektronischen, automatisierten Vorgänge in folgenden fünf Schritten ablaufen:

1. **Eingang** eines Dokuments, das den Vorgang auslöst und die nötigen Eingabeparameter für den Vorgang enthält. Identifizierung des Dokumenttyps, dadurch eindeutige Klassifizierung des Vorgangstyps. Bei unbekanntem Typ wird eine entsprechende Fehlerbehandlung durchgeführt.
2. **Transformation** des Dokuments in ein internes XML-Format
3. **Archivierung** des transformierten sowie des originalen Dokumentes in einem Dokumentenarchiv (Datenbank).
4. **Verarbeitung** des Dokumentes durch Weiterleiten an einen bestimmten Service. Dort Anbindung der Back-End-Systeme und deren Nutzung. Diese Aufgabe wird von einem Workflow-Management-System übernommen, das den zuständigen Service anhand von Regeln bestimmen kann. Dieser Schritt wird so lange wiederholt, bis das Dokument alle nötigen Bearbeitungsschritte durchlaufen hat. Alle Zwischenstufen des Dokumentes sowie das Endergebnis werden in dem Archiv gespeichert.
5. Schließlich Transformation in das gewünschte Ausgabeformat und **Ausgabe** des Dokuments.

Bei der Verarbeitung der Dokumente sollen verschiedenste Datenformate unterstützt werden. Für die Eingabe eines Dokumentes ist beispielsweise ein HTTP-Aufruf über das Internet oder eine E-Mail mit den standardmäßig bekannten aber auch proprietären Datenformaten denkbar. Bei der Ausgabe des Ergebnisdokuments wird eine flexible Unterstützung verschiedener Formate verlangt, um den Anwender in der Vielzahl möglicher Endgeräte nicht einzuschränken. Derzeit vorgesehen sind HTML-Seiten für Web-Browser, E-Mails, XML-Daten bis hin zu WML für WAP-fähige Handys. Man spricht in diesem Zusammenhang auch von *Multi-Channel-Lösungen*.

Die E-Business-Plattform steht nicht unabhängig von den existierenden Systemen da. Vielmehr soll sie eine Brücke zwischen den neuen Kommunikationsmöglichkeiten und den bestehenden Anwendungen schlagen. Zu den wichtigsten Anforderungen zählt daher eine gute Einbindung und Integration der bereits vorhandenen Back-End-Systeme und Softwarekomponenten.

Bei der Realisierung der Plattform sollen derzeitige und zukünftige Standards und Technologien berücksichtigt werden. Dies garantiert eine möglichst lange Lebensdauer des Systems und die Nutzung des aktuellen Stands der Technik. Explizit gefordert

wird in diesem Zusammenhang der Einsatz von Java und XML als Kerntechnologien. Java als Programmiersprache und XML als Grundlage der Datenformate eignen sich beide für heterogene Umgebungen, da sie sich durch hohe Plattformunabhängigkeit auszeichnen. Die E-Business-Plattform wird eine solche heterogene Umgebung darstellen, da verschiedene Systeme eingebunden werden müssen, die teilweise auch auf unterschiedlichen Hardware-Plattformen laufen. Mit XML lassen sich anwendungsneutrale Datenformate definieren, die zum Austausch von Informationen zwischen den einzelnen Komponenten der Plattform benutzt werden können. Mit der wachsenden Verbreitung des XML-Standards sind inzwischen auch eine umfangreiche Anzahl von Werkzeugen zur XML-Verarbeitung vorhanden, insbesondere für den Einsatz in Kombination mit Java-Software.

Nach der Übersetzung eines eingegangenen Dokuments in das interne XML-Format ist zur weiteren Verarbeitung ein Workflow-Management-System nötig. Über eine Sammlung von Regeln soll es feststellen, wem das Dokument als nächstes zugewiesen werden muss. Nach jedem Verarbeitungsschritt soll der neue Zustand des Dokuments in einem Archiv protokolliert werden. Dazu werden Kopien des Dokuments in einer eigenen Datenbank abgelegt.

Weitere, jedoch eher technische Anforderungen des Finanzunternehmens sind eine Verfügbarkeit des Systems rund um die Uhr mit gleichzeitig hoher Ausfallsicherheit. Außerdem sind Mechanismen zur Authentisierung und Autorisierung sowie zur Verschlüsselung kritischer Daten nötig, um ein ausreichendes Sicherheitsniveau zu erreichen. Die E-Business-Plattform soll im Zusammenhang mit einem erweiterten Internetauftritt durch den Aufbau eines sogenannten *Portals* eingeführt werden. Hierfür sind geeignete Werkzeuge zur Verwaltung der Informationsinhalte auf den Web-Seiten nötig. Dementsprechend soll ein Portalmanager in Verbindung mit einem Content-Management-System eingesetzt werden. Eine besondere Herausforderung ist dabei die *Personalisierung* der Inhalte, so dass jedem Anwender und Kunden eine individuell für seine Sicht zusammengestellte Informationsmenge angeboten werden kann.

3.1.3 Architekturvorschlag

Um die aufgestellten Anforderungen zu konkretisieren, wurde von der Informatikabteilung des Finanzinstituts ein erster Architekturvorschlag unterbreitet, der schematisch in folgender Abbildung wiedergegeben wird.

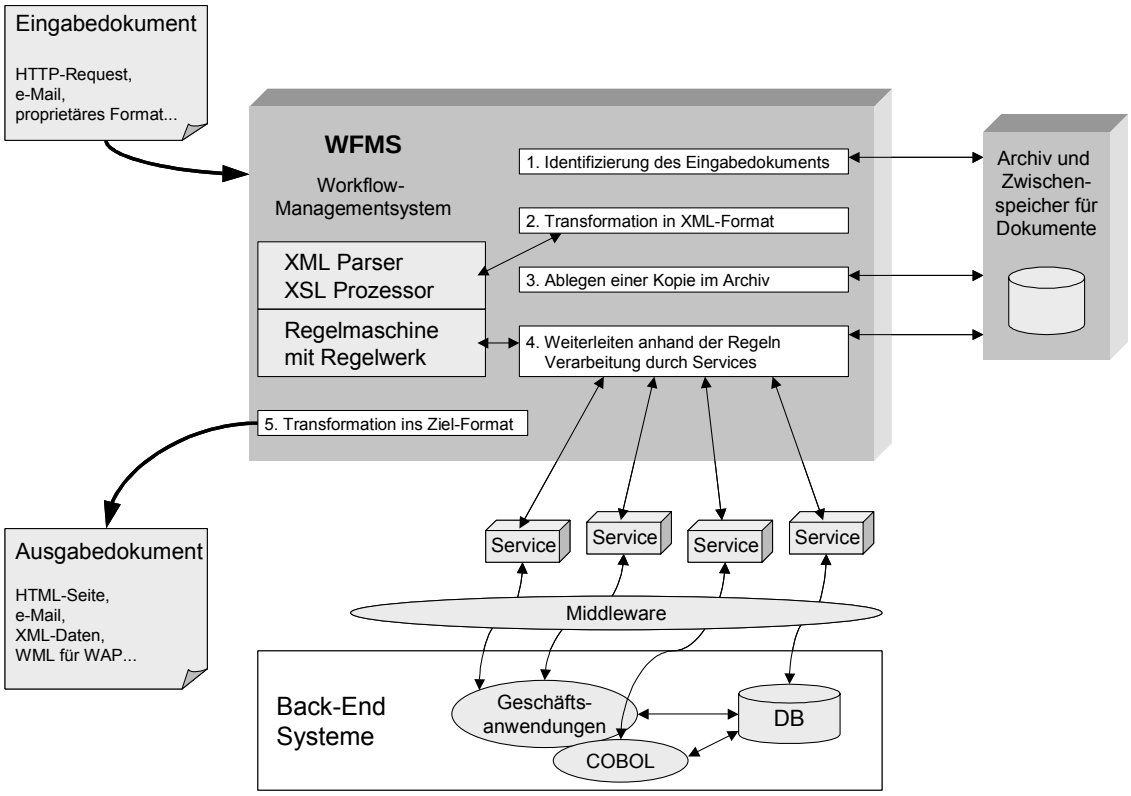


Abb. 3.1: Architekturvorschlag für eine E-Business-Plattform

Als zentrale Komponente der Plattform kann das Workflow-Management-System angesehen werden, welches die eingehenden Dokumente entgegennimmt und deren weitere Verarbeitung steuert, bis schließlich die entsprechende Antwort als Ausgabedokument auf die Anfrage geliefert wird. Dabei benutzt das WFMS andere Komponenten, um die Teilaufgaben der Formattransformation, der Archivierung, des Parsens und des Weiterleitens an spezielle Services mit Verbindung zu den Back-End-Systemen zu erfüllen. Das WFMS überwacht und steuert die automatisierte Verarbeitung des Geschäftsvorfalls bis zum Ende. Der Vorgang und sein augenblicklicher Zustand wird dabei jeweils durch das aktuelle Dokument repräsentiert.

Da das eingehende Dokument entsprechend den Anforderungen in sehr verschiedenen, inkompatiblen Formaten vorliegen kann, ist nach der Erkennung des Dokumentes zunächst eine Transformation in ein einheitliches, internes Format nötig. Als Grundlage für dieses Format wird XML benutzt. Werkzeuge wie ein XML-Parser und ein Stylesheet-Prozessor für die Transformationssprache XSLT (*eXtensible Stylesheet Language Transformation*, siehe [XSLT99]) helfen bei der Konvertierung in das interne Format. Sie werden auch nach der Verarbeitung benutzt, um die Dokumentdaten wieder in das vom Klienten geforderte Format zu konvertieren. Nicht eindeutig erkennbare oder falsch formatierte Dokumente werden ausgesondert und müssen manuell

überprüft werden. Daten, die bereits im richtigen Format ankommen, müssen nicht mehr transformiert werden. Für einen Ansatz zur Lösung der Transformationsprobleme im Zusammenhang mit XML und proprietären Formaten sei auf [Dep00] und die Studienarbeit [Lüt00] verwiesen. Eine XML-fähige Datenbank wird als Dokumentenarchiv eingesetzt. In ihr werden zur Protokollierung der Vorgänge Kopien der Dokumente gespeichert. Somit wird bei der internen Verarbeitung und Zwischenpufferung der Daten komplett auf XML gesetzt.

Das Weiterleiten (Routing) wird vom WFMS mit Hilfe einer Regelmaschine vorgenommen. So werden die in XML vorliegenden Dokumente intern an die richtigen Services weitergeleitet. Dabei dienen Tags der XML-Struktur zur Klassifizierung des Dokumenttyps und –zustands. Zu diesem Zweck hat das Unternehmen eine spezielle XML-Sprache für die interne Struktur der Dokumente definiert. Nach der Klassifizierung entscheidet das WFMS aufgrund der vorhandenen Regeln, welchem Service eine Referenz auf das Dokument gegeben werden soll, so dass sich der Service das Exemplar des Dokumentes aus der Dokumentdatenbank holen kann.

Die Services sollen alle in Java realisiert werden. Jeder Service ist ein Stellvertreter für eine Funktion der Back-End-Systeme und stellt dem WFMS eine solche Funktion zur Verfügung. Dazu bildet der Service eine Art Container für die Parameter, die zum Aufruf der entsprechenden Back-End-Funktion benötigt werden. Änderungen an den Funktionalitäten der Back-End-Systeme können möglicherweise die Erstellung neuer Services bedingen. Die Softwarekomponenten geben nach Ausführung ihrer Funktionen die Rückgabewerte an die Services zurück. Die Services erfüllen auf diese Weise die Aufgabe von Konnektoren zwischen dem WFMS und den existierenden Softwarekomponenten.

Der Routingmechanismus des WFMS beruht im Wesentlichen auf der Auswertung des Prozesszustandes und des Dokumenttyps durch ein Regelwerk. Für die Realisierung dieses Regelwerks kommen verschiedene Ansätze von einer einfachen Liste über ein XML-Dokument bis zur Datenbank in Frage. Es besteht aus einer endlichen Menge von Regeln, die alle möglichen Kombinationen der Parameter beschreiben und jeweils auf die zugehörigen Ausgabeparameter abbilden. Die Definition der Regeln passiert also auf relativ technischer Ebene und kann bei größeren Systemen schnell einen unüberschaubaren Umfang annehmen (vgl. Abschnitt 11.1). Da eine Regel im Prinzip wie eine Zustandsmaschine für einen Prozesszustand seinen nachfolgenden Zustand angibt, könnte man alternativ auch eine ablauforientierte Workflow-Modellierung vornehmen. Eine visuelle Modellierungssprache für Workflows, wie sie in Kapitel 6 vorgestellt wird, könnte bei der Spezifikation der Abläufe helfen.

Eine so aufgebaute E-Business-Plattform eignet sich, um die Vorgänge zur Verarbeitung von eingehenden Dokumenten automatisiert durchzuführen. Jedes Eingangsdokument stellt dabei eine Anfrage an die vorhandenen Systeme dar. Das WFMS ist über die Nutzung von Services bzw. Konnektoren in der Lage, die bestehenden Komponenten in die Verarbeitung zu integrieren. Eingebettet wird das WFMS in einen Rahmen von Systemen zur Anbindung an Kommunikationsnetze wie das Internet. Dazu werden ein Webserver sowie ein Applikationsserver vorgeschaltet, die alle Anfragen entgegennehmen und gegebenenfalls an das WFMS zur Verarbeitung weitergeben. Ein Portal-Management-System im Zusammenhang mit einem Content-Management-System zur Aufbereitung der Informationsinhalte runden die gesamte Architektur ab.

3.2 Anforderungen an Workflow-Systeme im E-Business

Im vorangegangenen Abschnitt wurden die Ideen und Konzepte für den Einsatz eines Workflow-Management-Systems innerhalb der E-Business-Plattform eines großen Finanzinstituts vorgestellt. Daraus ergeben sich spezielle Anforderungen, die ein solches System erfüllen muss. An dieser Stelle soll von den spezifischen Bedingungen der Fallstudie abstrahiert werden. Vielmehr soll eine allgemeine Liste der wichtigsten Anforderungen an Workflow-Management-Systeme formuliert werden. Zu diesem Zweck wurde die spezielle Situation der Fallstudie mit dem Workflow-Referenz-Modell aus [Wmc95] verglichen. Dadurch ergeben sich einige zusätzliche Forderungen, die in der Fallstudie nicht explizit aufgeführt wurden. Besondere Berücksichtigung bei den folgenden Betrachtungen soll das Anwendungsgebiet E-Business finden, in dem das Workflow-Management-System zum Einsatz kommt. Unter diesen Gesichtspunkten ergeben sich folgende Anforderungen:

1. Zunächst muss dem Benutzer ein visuelles *Modellierungswerkzeug* zur Verfügung stehen, mit dem er die vorhandenen oder gewünschten Prozessabläufe vollständig modellieren kann (siehe [Wmc95] S.12). Mit Hilfe dieses Werkzeugs kann er komfortabel auf grafischer Ebene Prozessmodelle entwerfen, ohne sie durch eine bestimmte textuelle Sprache – ähnlich einer Programmiersprache – beschreiben zu müssen.

2. Das Modellierungswerkzeug sollte den Benutzer durch geeignete Funktionen bei der *Analyse, Konsistenzprüfung und Optimierung* der Prozessmodelle unterstützen, da dies ein wichtiges Ziel der Workflow-Modellierung ist (vgl. [Geo95] S.136). Zu diesem Zweck bietet es sich an, eine *Simulation* von Workflow-Modellen durchzuführen, um ihr Verhalten schon vor dem realen Einsatz beobachten und untersuchen zu können.

3. Beim Einsatz eines Workflow-Management-Systems in einem Unternehmen ist davon auszugehen, dass dadurch keine vollständig neue Software-Lösung eingeführt werden soll. Vielmehr dient ein solches System häufig dazu, die heterogenen vorhandenen Komponenten bzw. Back-End-Systeme zu einer Gesamtlösung zu integrieren. Eine solche Situation ist auch bei der Fallstudie gegeben. Daher muss das *Einbinden von bestehenden Softwarekomponenten* sowohl bei der Modellierung als auch bei der Ausführung der Workflows möglich sein (vgl. [Böh95] S.15). Gerade diese Anforderung stellt laut [Geo95] (S.139) ein nicht triviales Problem dar.

4. Im Zusammenhang mit den Daten, die durch einen Prozess verarbeitet werden, unterscheidet das Workflow-Referenz-Modell zwei Arten (siehe [Wmc95] S.25). „Workflow-relevante Daten“ enthalten alle Informationen, die für die Ausführung einer bestimmten Prozessinstanz von Bedeutung sind. Sie werden beim Aufruf des Prozesses festgelegt bzw. übergeben und enthalten insbesondere Steuerungsinformationen, die den Prozessablauf direkt beeinflussen. Das Workflow-Management-System muss dafür sorgen, dass alle Komponenten *Zugriff auf die*

Workflow-relevanten Daten haben. Bei der Fallstudie können beispielsweise die Eingabedokumente, die jeweils einen Vorgang auslösen, als solche Daten interpretiert werden. Demgegenüber sind „Workflow-Applikations-Daten“ alle sonstigen Daten, die von den aktiven Komponenten des Systems gelesen oder geschrieben werden. Ihre Verarbeitung ist jedoch für das WFMS transparent.

5. Im Verlauf einer Prozessausführung müssen die Workflow-relevanten Daten maschinell verarbeitet werden. Mit der Integration bestehender Komponenten in einen Prozess entsteht zwangläufig das Problem des Datenaustausches zwischen heterogenen, inkompatiblen Systemen. In jüngster Zeit hat sich XML als Datenformat für heterogene Umgebungen stark verbreitet. Inzwischen wurden zahlreiche verwandte Sprachen im Umfeld von XML standardisiert. Sie dienen zur Anwendung und Verarbeitung von XML in bestimmten Branchen, zum Beispiel durch Transformationen von XML-Dokumenten. Mit der wachsenden Verbreitung sind außerdem eine Menge von Werkzeugen verfügbar, mit denen sich XML-Dokumente auf einfache Art verarbeiten lassen. Da sich XML somit als derzeitiger und auch zukünftiger Industriestandard etabliert hat, wird gerade im Bereich des E-Business verstärkt auf den Einsatz von XML-Technologien für die Datenhaltung und den Datenaustausch gesetzt. Auch die Anforderungen des Finanzunternehmens aus der in Abschnitt 3.1 vorgestellten Fallstudie schreiben explizit die Verwendung von XML vor. Aus diesen Gründen kann die *Verarbeitung von XML-Dokumenten* als eine Anforderung an heutige Workflow-Management-Systeme im Bereich E-Business angesehen werden.

6. Das Workflow-Management-System benötigt ein Werkzeug, mit dem sich aus dem grafischen Modell eines Prozesses eine *textuell codierte Beschreibung generieren* lässt, die so präzise ist, dass sie als Grundlage für die automatisierte Ausführung des Workflows dienen kann (vgl. [Geo95] S.136). Diese Workflow-Beschreibung muss in einem plattform- und anwendungsneutralen Format abgelegt werden, damit bei einer Änderung der Systemumgebung nicht alle bisherigen Modelle unbrauchbar werden. Aus diesem Grund sollte, ähnlich wie bei den Workflow-relevanten Daten, auch für die Prozessbeschreibung XML verwendet werden.

7. Damit ein Workflow-Modell maschinell durch das WFMS ausgeführt werden kann, muss es möglich sein, eine Zuordnung aller Modellelemente zu entsprechenden realen Komponenten zu definieren bzw. Teilsysteme zu identifizieren, die noch nicht existieren und daher neu entwickelt oder angeschafft werden müssen (vgl. [Geo95] S.129). Da diese Zuordnung nicht schon bei der Modellierung des Prozesses stattfinden muss, kann man auch von *dynamischer Bindung von Komponenten* sprechen.

8. Das WFMS muss eine Art Interpreter (Workflow Enactment Service, vgl. [Wmc95]) besitzen, der die *Kontrolle und Ausführung der Prozessinstanzen* leistet. Er greift dazu auf die formale Beschreibung des Workflow-Modells zurück und muss für die Ausführung aller anfallenden Aufgaben sorgen. Dazu zählt vor allem die Abarbeitung des Prozesses gemäß dem korrekten Kontrollfluss und das

Ansprechen der Back-End-Systeme sowie der Austausch von Daten mit diesen Systemen.

9. Im Anwendungsgebiet E-Business ist davon auszugehen, dass Geschäftsvorfälle und damit die Ausführung von Prozessinstanzen durch Benutzeraktionen an einer internetbasierten Oberfläche ausgelöst werden. Das WFMS sollte daher die notwendige Funktionalität besitzen, um ein *Web-Portal* als Benutzungsschnittstelle zu integrieren. Als besonders flexibel stellt sich ein sogenannter *Multi-Channel-Ansatz* dar, der die verschiedensten Ein- und Ausgabegeräte unterstützt. Dadurch kann das WFMS nicht nur mittels eines Web-Browsers, sondern beispielsweise auch per Handy angesprochen werden. Als Ausgabe eines Prozesses könnte neben HTML-Dokumenten auch WML, PDF usw. erzeugt werden.

10. Neben der Anbindung von Softwarekomponenten kann es in Workflow-Modellen auch notwendig sein, menschliche Bearbeiter während der Ausführung in den Prozessablauf einzubeziehen, um Teilaufgaben zu erledigen. Das WFMS sollte Mechanismen enthalten, mit denen eine solche *Benutzerinteraktion* realisierbar ist (Workflow Client Application Interface, vgl. [Wmc95] S.21).

11. Es sollte eine *Administrator-Schnittstelle* zur Verfügung stehen, über die die Ausführungskontrolle der einzelnen Prozessinstanzen möglich ist (siehe [Wmc95] S.44). Denkbar ist hier sowohl eine direkte Lösung in Form einer Benutzungsschnittstelle als auch eine API-Schnittstelle (Application Programming Interface).

12. Im Architekturvorschlag der Fallstudie ist ausdrücklich eine Regelmaschine geplant, die anhand einer Regelmenge entscheidet, welchem Service ein Eingabedokument mit einem bestimmten Vorgangstyp übergeben wird. Dabei handelt es sich um eine spezielle Anforderung, die nicht als allgemeingültig angesehen werden kann. Ein WFMS muss allerdings flexibel genug sein, um solche *Geschäftsregeln* umzusetzen, was aber nicht unbedingt in Form einer Regelmaschine erfolgen muss, sondern auch durch ablauforientierte Modelle mit bedingten Verzweigungen, wie etwa Zustandsmaschinen, realisiert werden kann (vgl. z.B. [Oes97]).

13. Ein Workflow-Management-System muss die Behandlung von Ausnahmezuständen vorsehen, damit das System stets in einen konsistenten Zustand verbleibt (vgl. [Böh95] S.13ff). Diese Forderung kann als Einhaltung von *Transaktionseigenschaften* bezeichnet werden. Dazu zählt beispielsweise das Wiederaufsetzen abgebrochener Prozesse nach einem Systemausfall oder das Rückgängigmachen von Prozessen, bei deren Ausführung ein Fehler aufgetreten ist. Zu diesem Zweck kann es notwendig sein, die Prozessabläufe zu protokollieren.

14. Das WFMS sollte *plattformunabhängig* sein, damit es nach möglichen Änderungen an der Systemumgebung weiterhin einsatzfähig bleibt.

3.3 Anforderungen an Workflow-Modellierungssprachen

Um den Modellierungsprozess für Workflow-Modelle systematisch zu unterstützen, bedarf es einer geeigneten Sprache, mit der ein Systemingenieur die Modelle spezifizieren kann. Im Zusammenhang mit den im vorherigen Abschnitt definierten Forderungen an Workflow-Management-Systeme bzw. deren Workflow-Modelle ergeben sich bestimmte Anforderungen, die eine geeignete grafische Modellierungssprache erfüllen muss.

Die Modellierungssprache sollte eine *visuelle Sprache* sein, damit man sich ein anschauliches Bild von den modellierten Prozessen machen kann. Eine textuelle Beschreibung der Prozesse in einer formalen Workflow-Sprache käme der Programmierung der Workflows gleich und widerspricht der Intention, die Aufgabe der Modellierung möglichst einfach zu gestalten. Laut [Aal97] erleichtert eine visuelle Modellierungssprache nicht nur das Lernen und Verstehen der Modelle, sondern verbessert auch die Möglichkeit, sie – unter Umständen sogar mit Laien – zu diskutieren. Aus diesem Grund ist es auch wünschenswert, dass zur Modellierung eine Sprache benutzt wird, die bereits *weitläufig bekannt* ist oder sich als Industriestandard etabliert hat, so dass mit größerer Akzeptanz bei den Anwendern gerechnet werden kann. Darum wurde in der vorliegenden Arbeit keine vollständig neue Sprache entworfen, sondern bestehende Modellierungssprachen mit den hier gestellten Anforderungen verglichen und auf ihre Tauglichkeit untersucht.

In [Wmc95] wird ein Geschäftsprozess als Folge von Aktivitäten definiert. Die auszuwählende Sprache sollte daher *ablauforientiert* sein, damit sie mit dieser Definition harmoniert. Zur Erfüllung dieser Forderung bietet sich die Anwendung von Sprachen an, die auf gerichteten Graphen basieren. Mit ihnen lassen sich Kontrollflüsse beschreiben. Als bekannte Vertreter dieser Gruppe von Modellierungsansätzen werden später Petri-Netze, Ereignisgesteuerte Prozessketten und UML-Aktivitätendiagramme näher vorgestellt (siehe Kapitel 5).

Des Weiteren sollen folgende Eigenschaften erfüllt werden:

1. Die Workflow-Management Coalititon (siehe [Wmc99] S.29ff) hat vier *grundlegende Kontrollflusskonstrukte* identifiziert, die von der Sprache unterstützt werden müssen:
   a) Sequenzen,
   b) bedingte Verzweigungen,
   c) Schleifen,
   d) Nebenläufigkeit bzw. parallele Teilprozesse und deren Synchronisation

2. Aufgrund der Definition eines Geschäftsprozesses als Folge von Aktivitäten mit einem gemeinsamen Geschäftsziel wird in [Aal97] die Forderung gestellt, dass jedes Workflow-Modell
   a) genau einen definierten *Startpunkt* und
   b) genau einen definierten *Endpunkt* besitzen muss.

3. Die Aktivitäten während der Ausführung des Workflows stellen in der Regel Operationen auf den Workflow-relevanten Daten in Form eines XML-Dokuments oder in Abhängigkeit von ihnen dar. Dazu muss die Modellierungssprache Konstrukte zum *gezielten Zugriff auf diese XML-Daten* bereitstellen. Damit wird es ermöglicht, den Kontrollfluss in Abhängigkeit von den vorhandenen Daten zu modellieren. Zusätzlich muss es möglich sein, im Modell Aussagen über die Struktur der XML-Daten an jeder Stelle des Prozesses zu machen. Dadurch können sowohl Fehler im Modell selbst, als auch Inkonsistenzen bei der späteren Ausführung des Prozesses besser erkannt werden.

4. Ein Workflow-Modell besteht entsprechend der Definition (vgl. [Wmc95]) aus einer *Verkettung von Aktivitäten*. Eine Aktivität in den hier betrachteten XML-basierten Workflow-Modellen repräsentiert eine bestimmte Softwarekomponente, die Transformationen auf den XML-Daten und gegebenenfalls andere Aktionen wie das Versenden von Nachrichten oder Datenbankbefehle ausführt. Mit der Modellierungssprache müssen solche Aktivitäten in den Workflow eingefügt werden können. Transitionen modellieren den Kontrollfluss zwischen den einzelnen Aktivitäten.

5. Wenn bestehende Applikationen in einen Workflow integriert werden, müssen bei ihrem Aufruf „komplexe Startparameter“ übergeben werden. (siehe [Böh95] S.15). Die Modellierungssprache muss es daher ermöglichen, *Parametereinstellungen* für eine Aktivität bzw. die mit ihr korrespondierende Softwarekomponente anzugeben. Die den Parametern zugewiesenen Werte sind entweder statisch oder aus den verfügbaren XML-Daten entnommen. Dazu muss auf eine Stelle im XML-Dokument verwiesen werden können.

6. Einer Aktivität kann auch ein schon definierter Workflow zugeordnet werden. Somit muss es mit der Modellierungssprache möglich sein, die Prozesse in einer *hierarchischen Struktur* zu schachteln. Diese Möglichkeit sorgt nicht nur dafür, dass die Modelle von komplexen Prozessen übersichtlich und beherrschbar bleiben (vgl. [Aal97]), sondern ermöglicht auch das Einbinden von bereits vordefinierten Elementarprozessen. Beim Erreichen einer solchen Aktivität bei der Ausführung des Workflows wird der Teil-Workflow wie ein Makro aufgerufen.

7. Die Akteure innerhalb eines Workflow-Modells, d.h. Softwarekomponenten und Sub-Workflows, sollen durch ein *statisches Modell* repräsentiert werden (vgl. dazu Abschnitt 2.3.1). Eine Modellierungssprache sollte geeignete Konstrukte zur grafischen Darstellung eines solchen Modells besitzen.

8. Bei der Ausführung von systemorientierten Workflows muss häufig die Konsistenz der beteiligten Daten sichergestellt werden (siehe [Böh95] S.14). Die Modellierungssprache sollte daher ein Konstrukt enthalten, mit dem sich Anfang und Abschluss von *Transaktionen* definieren lassen. Eine Transaktion ist dabei als Teil eines Workflows anzusehen, der atomar abläuft, also entweder ganz von Anfang bis Ende oder gar nicht (vgl. Abschnitt 2.2.2).

9. Mit einem *Zeitmodell*, wie es in [Geo95] (S.135) und [Böh95] (S.8) für die Modellierungssprachen gefordert wird, können Angaben über die Dauer von Aktivitäten sowie ihre Start- und Endzeitpunkte gemacht werden. Durch diese Spezifikation kann ein WFMS die Terminvorgaben überwachen und bei deren Verletzung entsprechende Warnungen ausgeben oder Gegenmaßnahmen ergreifen.

3.4 Zielsetzungen dieser Arbeit

Nachdem in den vorausgegangenen Abschnitten auf die Anforderungen an Workflow-Management und insbesondere die Modellierung von Workflows eingegangen wurde, wollen wir hier die Zielsetzungen der Arbeit verdeutlichen und dabei eine Gewichtung der Anforderungen vornehmen.

Das Hauptziel ist die Konzeption eines durchgängigen Verfahrens zum Management von Workflows von der Modellierung bis zur Ausführung. Neben den Konzepten soll ebenso die Umsetzung in entsprechende Software und Werkzeuge durchgeführt werden. Da es sich beim Workflow-Management um ein überaus weitläufiges Forschungsgebiet handelt, wollen wir uns in dieser Arbeit auf die Anforderungen konzentrieren, die sich im Zusammenhang mit E-Business-Plattformen ergeben. Das resultierende WFMS ist daher besonders für den Einsatz im E-Business und das Management der dort anfallenden Workflows konzipiert. Bei diesen Workflows liegt der Schwerpunkt sehr stark auf rein systemorientierten Prozessen zur Verarbeitung von Daten und Informationen. Es geht nicht um die Koordinierung der menschlichen Zusammenarbeit in einem Unternehmen, die durch eine Vorgangssteuerung, wie man sie bei der sogenannten Groupware findet, unterstützt werden soll. Aus diesem Grund können wir die Problematik der Benutzerinteraktion bei unseren Betrachtungen weitgehend ausklammern. Das Resultat sollen Workflows sein, die vollständig automatisiert ablaufen und die auf der E-Business-Plattform aufgerufenen Funktionen – zum Beispiel von einem Kunden aus einem Web-Portal heraus – abarbeiten können. Daher ist eine Anbindung des WFMS an ein Portal vorzusehen.

Die Aufgabenträger der einzelnen Aktivitäten sind also nicht Mitarbeiter eines Betriebes, sondern Software-Einheiten und existierende Back-End-Systeme. Diese Komponenten sollen durch den Workflow konfiguriert und in die E-Business-Umgebung integriert werden. Dazu ist ein ausreichendes Maß an Interoperabilität der bestehenden Systeme mit dem WFMS und der E-Business-Plattform erforderlich. Zur Kopplung der Komponenten müssen mit ihnen Daten ausgetauscht werden, die in einem anwendungsneutralen Format vorliegen müssen. In dieser Arbeit wollen wir dazu den derzeitigen Standard XML zu Grunde legen und untersuchen, wie sich XML-Konstrukte und -Technologien in das Workflow-Management integrieren lassen. In diesem Zusammenhang wird davon ausgegangen, dass ein Workflow seine Workflow-relevanten Daten durch Eingabe eines XML-Dokumentes erhält und sein Ergebnis ebenfalls als XML-Dokument zurückliefert. Wenn diese Verarbeitung der Dokumente intern durch den Fluss des Dokumentes durch die einzelnen Verarbeitungsaktivitäten realisiert wird, wollen wir im Folgenden von einem „XML-Prozess“ sprechen. Für die Konzeption und Realisierung von XML-Prozessen sei auf die Kapitel 6 und 7 verwiesen.

Ein besonderes Augenmerk wird auf der Modellbildung von Workflows liegen. Insbesondere sollen schon bei der Modellierung die Eigenheiten von XML-Prozessen, das sind vor allem die Einbindung von Softwarekomponenten und die Verarbeitung von XML-Dokumenten, berücksichtigt werden. Darum muss eine Modellierungssprache gefunden werden, mit der sich Modelle der XML-Prozesse als sogenannte *Prozessdiagramme* beschreiben lassen (vgl. Kapitel 6). Die Prozessdiagramme sollen von ihrer Semantik her so gut definiert sein, dass sie sich ggf. nach automatisierten Konvertierungen als Eingabe für eine maschinelle Ausführung der XML-Prozesse durch das Workflow-Management-System eignen. Somit ist es ein wichtiges Ziel, die Implementierung der Workflows und die damit verbundene Konfiguration der Komponenten auf die Erstellung eines graphischen Modells zu beschränken. Alles übrige kann von den Werkzeugen automatisiert erledigt werden.

Die gewählte Modellierungssprache muss die im Abschnitt 3.3 gestellten Anforderungen erfüllen. Lediglich die Forderung nach einem Zeitmodell ist bei der Modellierung der systemorientierten Informationsprozesse nicht von entscheidender Wichtigkeit. Aussagen über Starttermine wie „frühestens am …“ oder Endtermine wie „spätestens bis zum …“ und die damit vom WFMS zu leistende termingerechte Vorlage des Auftrags beim zuständigen Bearbeiter finden sich vor allem bei interaktiven Prozessen mit menschlichen Sachbearbeitern (vgl. [Böh95] S.8). Bei den hier zu betrachtenden systemorientierten Workflows ist es lediglich ein Ziel, die gesamte Ausführungszeit zu minimieren. Aus den praktischen Anforderungen der Fallstudie ergeben sich keine Anzeichen für einen dringenden Bedarf an einer ausführlichen Zeitmodellierung. Aus diesen Gründen werden wir uns darauf beschränken, Zeitmodelle bei Modellierungssprachen nur zu betrachten, wenn sie bereits vorhanden sind, aber keine neuen Ansätze in dieser Richtung vorstellen.

Die sich aus der Modellierung ergebenden Prozessdefinitionen sollen ebenfalls in einem XML-Format abgelegt werden können (vgl. Kapitel 8). Dies ist neben der Plattformneutralität der XML-Sprachen besonders wegen der Kompatibilität von XML zu gängigen Kommunikationsprotokollen im Internet von Vorteil, weil sich so Prozessbeschreibungen über das Netz leicht austauschen lassen. Für eine solche XML-basierte Prozessbeschreibung muss eine ausreichend formale Definition gefunden werden, damit das WFMS die Beschreibung zwecks Ausführung interpretieren kann.

Der Entwicklung von komfortablen Hilfswerkzeugen zur Analyse und Optimierung der Workflow-Modelle schenken wir aus Zeitgründen keine Beachtung. Gleiches gilt für eine umfangreiche Schnittstelle zur Administration laufender Prozesse. Diese Anforderungen (siehe Punkte 2 und 11 in Abschnitt 3.2) können prinzipiell später durch entsprechende Ergänzungen erfüllt werden. Für die Anforderung der Zusicherung von Transaktionseigenschaften und zur Behandlung von Ausnahmen nach Fehlerzuständen können in dieser Arbeit wegen des Umfangs und der Schwierigkeit dieses Problemfeldes nur entsprechende Konzepte angedacht, aber nicht tiefergehend behandelt werden (vgl. Abschnitt 10.3 und 10.4). Trotzdem soll jedoch zumindest die spätere Ergänzung durch vorausschauenden Entwurf des Modellkonzepts offen gehalten werden.

Das prinzipielle Vorgehen in den nachfolgenden Kapitel gestaltet sich derart, dass wir durch die Vorstellung einiger exemplarischer Ansätze einen Einblick in den aktuellen Stand der Technik geben möchten. Die jeweiligen Verfahren, Modelle und Systeme werden anhand der hier gestellten Anforderungen bewertet, bevor wir unsere eigenen Ansätze oder Ergänzungen der bisherigen Konzepte vorstellen.

4 Evaluierung vorhandener Workflow-Systeme

Im Bereich der Workflow-Modellierung und -Ausführung gibt es eine Vielzahl von Lösungen und Produkten sowohl aus dem kommerziellen als auch aus dem wissenschaftlichen Bereich. Um einen Einblick in den aktuellen Stand der Technik zu gewähren, sollen an dieser Stelle exemplarisch drei Systeme vorgestellt werden, die den Zielen dieser Arbeit in besonderer Weise nahe kommen. Neben der jeweiligen Systemarchitektur wollen wir besonders die Aspekte der Workflow-Modellierung, Workflow-Ausführung und der Komponentenintegration betrachten. Im letzten Abschnitt dieses Kapitels findet eine zusammenfassende Evaluierung der Systeme anhand der Forderungen aus Kapitel 3 statt.

Zum einen handelt es sich um das kommerzielle, von Microsoft angebotene System *BizTalk Server 2000*, das zur Umsetzung der *BizTalk Initiative* beitragen soll. Die Betrachtung von BizTalk ist deswegen so interessant, weil es bei der Modellierung und Ausführung von Geschäftsprozessen in besonderem Maße auf den Einsatz von XML-Technologien setzt.

Zum anderen wird die von M. Brian Blake unter dem Namen *WARP* veröffentlichte, agentenbasierte Architektur zur Konfiguration verteilter Komponenten vorgestellt. In diesem Ansatz wird zwar nicht mit XML gearbeitet, er zeigt aber gut, wie man verteilte Softwarekomponenten durch Workflow-Systeme integrieren kann.

Schließlich folgt eine Erläuterung der grundlegenden Konzepte des Produkts *sunShine* der Firma S&N AG. Dabei handelt es sich um eine Integrationsplattform für B2C- und B2B-Anwendungen, wie beispielsweise Online-Banking-Systeme. Mit Hilfe von sunShine als Rahmenwerk lässt sich neben dem Layout auch die Businesslogik solcher Webauftritte realisieren.

Die Untersuchung von BizTalk Server 2000 und WARP erfolgte anhand von vorliegender Literatur und Dokumentation, bei sunShine war es dank der Kooperation mit der S&N AG möglich, Erfahrungen am installierten System selbst zu sammeln.

4.1 Microsoft BizTalk Server 2000

Die von Microsoft initiierte *BizTalk-Initiative* beschäftigt sich mit dem Problem, dass die Integration von Softwaresystemen verschiedener Unternehmen oder Unternehmensteile oft durch inkompatible Datenformate gebremst wird. Für die zum Austausch von Nachrichten erforderlichen abgestimmten Nachrichtenformate steht als Basistechnologie zwar XML zur Verfügung, es mangelt aber häufig an der Verständigung auf gemeinsame Schemata und Definitionen für die Nachrichtenformate (siehe [Mic00] S.3).

Das BizTalk-Konzept von Microsoft besteht im Wesentlichen aus den drei Teilen *BizTalk Framework*, der Internetplattform *BizTalk.org* und den *BizTalk Servers*. Ziel ist es, mit diesen Systemen eine Infrastruktur für unternehmensinterne wie -übergreifende Integration von Anwendungen und Geschäftsprozessen durch den Austausch von Daten in wohldefinierten und unter den Teilnehmern abgestimmten XML-Formaten anzubieten.

BizTalk Framework

Das BizTalk Framework definiert Regeln zur Erzeugung von BizTalk-konformen XML-Schemata, die die Struktur von austauschbaren Nachrichten festlegen (vgl. [Moh01] S.14f). Das Konzept basiert auf dem XML-Standard und dem offenen Nachrichtenaustauschprotokoll SOAP (Simple Object Access Protocol, siehe [SOAP01]), das vom W3-Konsortium zum Informationsaustausch und Funktionsaufruf in heterogenen, verteilten Umgebungen standardisiert worden ist. Damit ist es nicht mehr nötig, für die verschiedenen Anwendungen Interoperabilität durch gemeinsame Protokolle, gleiche Programmiersprachen, gleiche Betriebssysteme und ähnlichem herzustellen. Dieser Ansatz scheitert allein durch die Heterogenität der eingesetzten Systeme. Die Zusammenarbeit der Applikationen soll bei BizTalk vielmehr auf dem Aufruf von Funktionen durch das Erzeugen und Versenden einer XML-Nachricht beruhen. Der Empfänger – in diesem Fall ein BizTalk Server – muss die Nachricht lediglich verstehen, die nötigen Aktionen ausführen und gegebenenfalls seinerseits mit einer Nachricht antworten. Grundsatz ist somit eine Integration durch lose gekoppelte Kommunikationsprozesse.

Im Einzelnen definiert das BizTalk Framework eine Reihe von XML-Elementen, die sogenannten *BizTags* (siehe [Moh01] S.15), die für den Einsatz der XML-Formate bei der Anwendungsintegration und dem Nachrichtenaustausch nützlich sind. BizTalk gibt zum Beispiel XML-Tags vor, die Informationen für das Versenden von Nachrichten zwischen den betrieblichen Anwendungssystemen enthalten, die sogenannten „message elements“. Ein Dokument, das mit diesen speziellen Marken angereichert ist, stellt eine Erweiterung eines SOAP-Dokuments dar und wird *BizTalk-Dokument* genannt. Inzwischen hat Microsoft die Festlegung und Definition der BizTalk XML-Elemente an das Internet-Standardisierungsgremium W3C übergeben (siehe [Biz01]). Ein registrierter Benutzer kann nach der BizTalk Framework Spezifikation ein gültiges XML-Schema entwerfen und dieses BizTalk-konforme Format auf der BizTalk.org Plattform veröffentlichen, um es seinen Geschäftspartnern bekannt zu machen.

BizTalk.org

Mit der Webseite BizTalk.org [Biz01] wird ein für jedermann nach einer Registrierung zugänglicher Raum geschaffen, in dem man sich auf gemeinsame XML-Formate verständigen und diese austauschen kann. Diese zentralisierte Speicherung der XML-Formatdefinitionen in einem Schema-Repository soll den Austausch vereinfachen. Für die Suche nach einem bestimmten Schema können Stichworte angegeben werden, nach denen die Schema-Einträge kategorisiert sind. Nachdem ein Unternehmen die von einem Handelspartner benutzten Schemata gefunden hat, kann es entscheiden, ob es mit den gleichen Schemata arbeiten oder lieber eine Konvertierung der Nachrichten in die eigenen Dokumentformate vornehmen möchte (vgl. [Moh01] S.16f).

Ein auf BizTalk.org veröffentlichtes Schema ist auf Korrektheit geprüft und zu dem BizTalk Framework konsistent, so dass ein BizTalk Server Nachrichten dieses Formats interpretieren kann. Zusätzlich kann sich ein Interessent für ein bestimmtes Schema registrieren lassen, um automatisch Informationen über neue Versionen und Änderungen zu erhalten. Neben der Speicherung der XML-Formate dient die Internetplattform BizTalk.org zum Austausch von Neuigkeiten und Informationen über den jeweils aktuellen Stand der BizTalk-Initiative.

BizTalk Servers

Ein BizTalk Server ist ein Softwaresystem, das BizTalk Dokumente verarbeiten kann. Insbesondere muss der Server die im BizTalk Framework definierten BizTags interpretieren und gemäß der im Framework definierten Semantik verarbeiten können. Da es sich bei dem Framework um einen offenen Standard handelt, kann er prinzipiell von verschiedenen Anbietern implementiert und in ihre Softwaresysteme integriert werden. Zur Zeit arbeiten mehrere Unternehmen an entsprechenden Realisierungen (siehe [Moh01] S.18). Das augenblicklich einzige fertig gestellte und auch bekannteste Produkt in diesem Bereich ist der von Microsoft entwickelte BizTalk Server 2000, der im Folgenden näher vorgestellt werden soll.

4.1.1 Systemarchitektur von BizTalk Server 2000

Microsofts BizTalk Server 2000 erlaubt neben der Interpretation von BizTalk Dokumenten und Nachrichten auch die Entwicklung komplexer Geschäftsprozessmodelle und deren Ausführung. Das dabei eingesetzte Workflow-Management-System basiert auf dem Austausch von BizTalk Nachrichten und dem damit verbundenen Aufruf von Funktionen anderer, gegebenenfalls auch externer Systeme. Der Fortschritt der Abläufe kann über Wochen und Monate hinweg verfolgt werden. BizTalk Server will damit langlebige, lose gekoppelte Flüsse in verteilten Umgebungen wie einer unternehmensübergreifenden Wertschöpfungs- oder Lieferkette unterstützen.

Obwohl das System Anwendungen auf verschiedenen Plattformen integrieren soll, ist es selbst nicht plattformunabhängig. Die Systemvoraussetzungen zum Einsatz des BizTalk Servers sind vielmehr stark von Microsoft Basisprodukten wie dem Betriebssystem Windows 2000 geprägt (siehe [Mic01]).

Wie in [Moh01] S.19 dargestellt, handelt es sich bei BizTalk Server 2000 im Wesentlichen um eine Sammlung verschiedener Werkzeuge zur effizienten Anwendungsintegration: Zur Workflow-Modellierung kann man das grafische Werkzeug *Orchestration Designer* benutzen. Die damit erstellten Ergebnis-Modelle werden in das XML-Format *XLANG* übersetzt und vom *XLANG-Scheduler* ausgeführt. Der Scheduler ruft bei der Durchführung die *BizTalk Messaging Services* auf. In Verbindung mit einem *Messaging Manager* wird damit der Austausch von XML-Nachrichten und BizTalk-Dokumenten als Basiskonzept des BizTalk Servers realisiert. Als Administrationsschnittstelle zur Verwaltung der Server-Eigenschaften und der laufenden Prozessinstanzen dient das *Server Administration* Tool. Die Verfolgung einzelner Nachrichten kann mit dem *Document Tracking* geschehen. Des Weiteren gibt es einen *Editor* zur visuellen Definition der Nachrichtenformate und einen *Mapper* zur Definition von Abbildungen zwischen zwei verschiedenen Dokumentformaten. Diese Abbildung kann mit Hilfe von XSLT automatisch in eine Konvertierungsroutine umgesetzt werden.

BizTalk Server 2000 bietet keine Integration eines Internet-Portals, mit dem – wie in Abschnitt 3.2 gefordert – die Prozesse ausgelöst und ihre Resultate präsentiert werden könnten. Insbesondere wird in diesem Zusammenhang kein Multi-Channel-Ansatz verfolgt, um verschiedene Typen von Endgeräten bedienen zu können.

4.1.2 Komponentenintegration mit BizTalk Server 2000

Durch das Versenden von BizTalk-Dokumenten und Nachrichten spricht der BizTalk Server andere Komponenten und externe Systeme an. Dafür lassen sich die vorhandenen Nachrichtendienste mit den Aktionen des Workflows koppeln. Die in [Mic00] S.7 vorgestellte *Any-To-Any-Integration* soll über Adaptoren zu COM-Komponenten, Skript-Komponenten und zu Anwendungen Dritter realisiert werden. Des Weiteren lassen sich durch Einsatz des SOAP-Protokolls andere Komponenten aufrufen, die dieses Protokoll unterstützen. Mit diesen Schnittstellen möchte Microsoft ihren Server als Integrationsplattform für heterogene Umgebungen platzieren.

Unter dem Stichwort der *dynamischen Prozesse* ist es möglich, abstrakte Stellvertreter für Komponenten oder Organisationseinheiten zu definieren und zunächst nur diese in einem Workflow-Modell zu benutzen. Erst zur Laufzeit erfolgt dann die Auswahl einer konkreten Instanz oder Implementierung der Komponente, die dann jedes Vorkommen des abstrakten Stellvertreters ersetzt (vgl. [Mic00] S.7). Die in den Anforderungen aus Abschnitt 3.2 geforderte dynamische Bindung von Komponenten wird dadurch also erfüllt.

4.1.3 Workflow-Modellierung mit BizTalk Server 2000

Das in BizTalk Server integrierte grafische Modellierungswerkzeug für Workflows ist der Orchestration Designer (vgl. [Moh01] S.27ff), der auf Microsoft Visio basiert. Das Werkzeug erlaubt die visuelle Definition der Prozessmodelle und der Bindung der Aktionen an vorhandene Softwarekomponenten, jedoch nicht darüber hinausgehende Analysefunktionen zur Optimierung und Konsistenzprüfung.

Zunächst wird bei der Modellierung ein Geschäftsprozess durch ein Flussdiagramm dargestellt (siehe [Moh01] S.27ff), dessen Notation an die von UML-Aktivitätendiagrammen angelehnt ist. Die Modellierungssprache unterstützt Konstrukte für Sequenz, Verzweigung, Nebenläufigkeit und Synchronisation (siehe [Mic00] S.8) und erlaubt damit die Definition eines flexiblen Workflow-Modells. Des Weiteren lassen sich Teilbereiche des Prozessmodells zu einer Transaktion zusammenfassen, die atomar ausgeführt werden soll. Beim Entwurf der Workflows wird die Definition des Prozesses explizit von der zu Grunde liegenden Software getrennt. In einer Art statischem Modell werden die vorhandenen Softwarekomponenten aufgeführt und mit den einzelnen Prozessaktivitäten im dynamischen Modell verknüpft (siehe [Mic00] S.6). Dazu muss angegeben werden, welcher Nachrichtenfluss zwischen den Aktionen bzw. Komponenten gewünscht wird. Außerdem ist es möglich, einen Prozess als Komposition von mehreren Teilprozessen darzustellen (siehe [Mic00] S.7), um dadurch die Komplexität zu reduzieren und arbeitsteiligen Entwurf zu unterstützen.

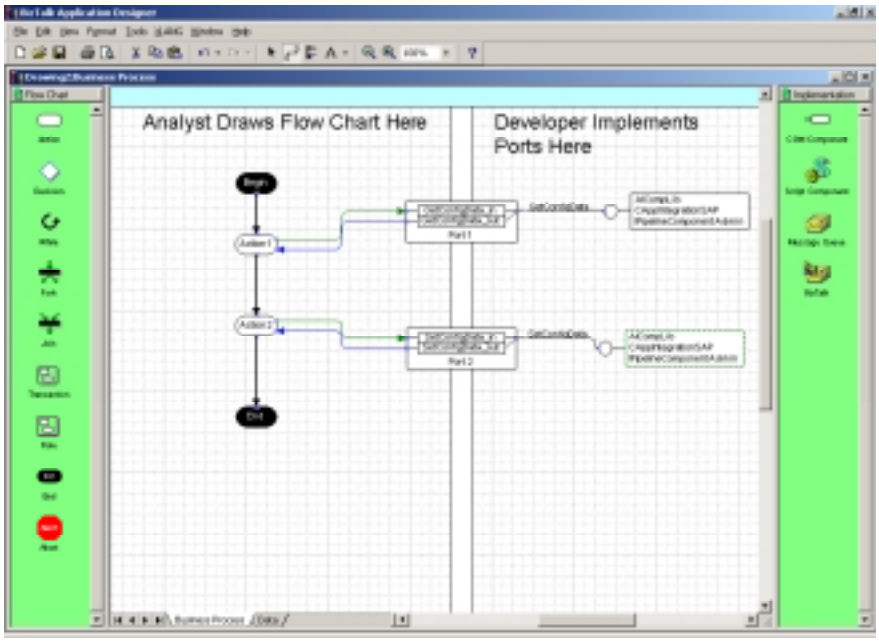


Abb. 4.1: Ansicht des BizTalk Orchestration Designer (aus: [Mic00] S.6)

Nach der Fertigstellung eines Workflow-Modells wird das Modell in das XML-Format XLANG umgewandelt und abgespeichert (vgl. auch Abschnitt 8.2.1). Damit liegt eine Workflow-Beschreibung in XML vor, die als sogenanntes *XLANG-Schedule* von dem BizTalk XLANG-Scheduler ausgeführt werden kann.

4.1.4 Workflow-Ausführung mit BizTalk Server 2000

Bei der Ausführung von Workflows spielt der XLANG-Scheduler eine zentrale Rolle. Die *XLANG Scheduler Engine* führt die einzelnen Instanzen eines durch einen XLANG-Schedule definierten Workflows aus. Der Start eines Ablaufs wird dabei durch den Aufruf einer Windows-spezifischen URL, dem sogenannten *moniker*, ausgelöst (siehe [Moh01] S.29). Bei der Ausführung greift der Scheduler auf die serverinternen Messaging Services zurück, um den notwendigen Nachrichtenaustausch zu realisieren. Außerdem verwaltet er die Ressourcen für die Instanzen, was besonders wichtig wird, wenn es sich um langlebige Prozesse handelt, die über viele Tage laufen können und eine große Anzahl solcher Prozesse aktiv ist. Muss ein Prozess lange auf eine bestimmte Nachricht warten, wird er im aktuellen Zustand eingefroren, bis die Nachricht eingetroffen ist. In dieser Zeit können die Betriebsmittel von anderen Instanzen genutzt werden (vgl. [Moh01] S.28).

Das Workflow-Konzept von BizTalk Server ist sehr stark datenorientiert und fokussiert die Kommunikation per Austausch von XML-Nachrichten. Die dafür benutzten Messaging Services verwenden BizTalk Dokumente als XML-Container, um neben XML auch andere Datenformate darin zu speichern; insbesondere wird EDI (Electronic Data Interchange), der bisherige Standard beim geschäftlichen Nachrichtenaustausch, unterstützt (siehe [Moh01] S.21). Empfängt BizTalk Server eine solche Nachricht, leitet er sie dem Nachrichtentyp entsprechend an eine bestimmte Organisationseinheit oder Komponente weiter. Damit ist eine flexible Verarbeitung von XML-Dokumenten möglich, und durch den Nachrichtenaustausch werden die einzelnen Komponenten gleichzeitig mit den nötigen Workflow-relevanten Daten versorgt.

Über die Konzepte *Port* und *Channel* werden für die einzelnen sendenden wie empfangenden Komponenten Eigenschaften wie zum Beispiel das zu Grunde liegende Protokoll festgelegt, die für einen reibungslosen Nachrichtenaustausch benötigt werden. Die Nachrichtenübertragung erfolgt entweder per Windows-spezifischer Kommunikation oder über offene Internetprotokolle wie SMTP und HTTP (siehe [Moh01] S.22ff). Warteschlangen-Konzepte erlauben das Puffern von Nachrichten, was die Zuverlässigkeit und Skalierbarkeit des Systems verbessert (vgl. [Moh01] S.26).

Wird bei der Modellierung ein Prozessteil als Transaktion gekennzeichnet, ist dafür ein Kompensationsprozess zu beschreiben, der ausgeführt werden soll, wenn der Prozessteil aufgrund eines Fehlers scheitert oder nicht vollständig ausgeführt werden konnte (vgl. [Mic00] S.9). Dieses Konzept ermöglicht zwar eine recht einfache Umsetzung einer Transaktionsunterstützung, wünschenswert wäre aber eine automatische Rücksetzung eines abgebrochenen Prozesses ohne jeweilige Angabe einer Kompensationsprozedur.

In den einzelnen Aktionen des Workflows werden vorhandene Softwarekomponenten aufgerufen. Die Einbindung von Benutzerinteraktionen ist dabei nicht explizit vorgesehen; sie kann aber sicher dadurch realisiert werden, dass eine aufgerufene Komponente intern mit einem Benutzer interagiert.

4.2 Komponentenkonfiguration mit WARP

Der von Brian Blake in [Bla00] vorgestellte Ansatz *WARP (Workflow Automation through Agent-based Reflective Processes)* behandelt ein agentenbasiertes System zur Konfiguration und Integration von verteilten Softwarekomponenten durch die Modellierung und Ausführung von Workflows. Blake sieht sein Konzept im Kontext der sich stark fortentwickelnden Komponententechnologie, die seiner Meinung nach große Auswirkungen auf die zukünftigen Softwareentwicklungsprozesse haben wird. Demnach wird es in Zukunft möglich sein, Software einfach durch Anordnung und Konfiguration existierender Komponenten – ggf. als Softwarebausteine von Dritten angeboten – in einer bottom-up Vorgehensweise zusammenzusetzen (vgl. [Bla00] S.1). Ein zentraler Vorteil ist dabei die vermehrte Wiederverwendbarkeit der einzelnen Java Beans, Klassen und Module, die jeweils Teilaufgaben übernehmen. Eine gute Möglichkeit, diese Bausteine miteinander zu verbinden, ist die Definition eines Workflows, der von autonomen Agenten ausgeführt werden kann. Mit WARP hat Blake ein solches Agentensystem vorgestellt.

Bevor die Prozesse von den Agenten ausgeführt werden können, müssen zwei Schritte vorausgehen: Zunächst müssen die verfügbaren Komponenten bzw. Dienste analysiert werden. Nachdem bekannt ist, welche Dienste zur Verfügung stehen, muss ein Workflow-Designer eine Spezifikation des Prozesses als objektorientiertes UML-Modell aufstellen. Dies kann dann den Agenten übergeben und von ihnen ausgeführt werden.

4.2.1 Systemarchitektur von WARP

Die Architektur des Systems besteht aus zwei Schichten: der Schicht zur Konfiguration des Systems und Spezifikation neuer Prozesse *(Automated Configuration Layer)* sowie die Schicht zur Ausführung der Workflows und zur Steuerung der vorhandenen Dienste

(Application Coordination Layer). Die folgende Abbildung zeigt die WARP-Architektur im Überblick:

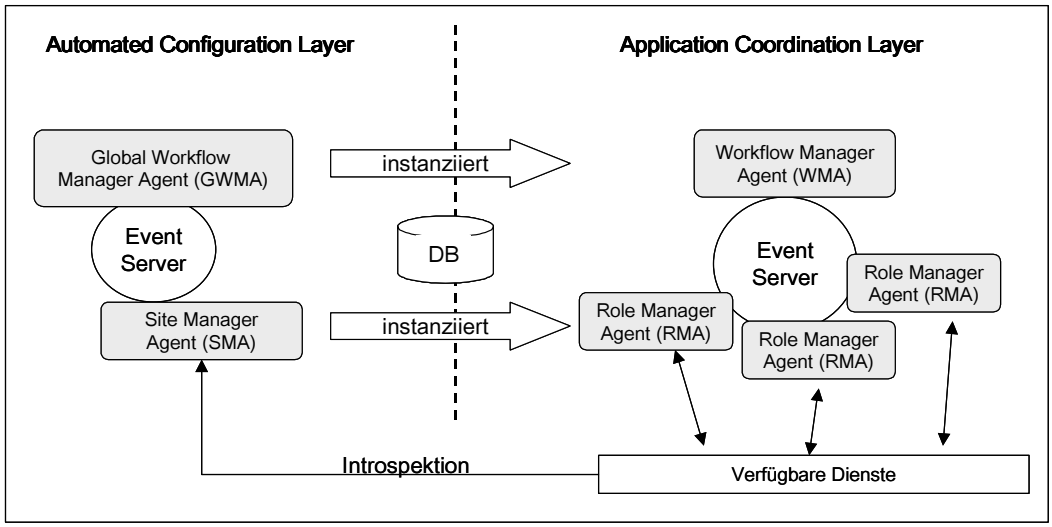


Abb. 4.2: WARP Architektur (aus [Bla00])

Die in der Konfigurationsschicht tätigen Agenten interpretieren neue oder geänderte Prozessspezifikationen und erzeugen dazu passende Instanzen der ausführenden Agenten in der Anwendungsschicht. Als gemeinsamer Datenspeicher der verschiedenen Agenten dient eine Datenbank. Aus den dort abgelegten Konfigurationsinformationen zu einem Workflow können sich die Agenten der Anwendungsschicht selbst konfigurieren, bevor sie die Ausführung des Workflows und die Kommunikation mit den Softwarediensten übernehmen. Neben der Datenbank existiert ein sogenannter Event Server. Dabei handelt es sich um einen für alle zugänglichen Punkt zur Kommunikation zwischen den verschiedenen Agenten. Jeder Agent kann dort bei Bedarf Ereignisse und Nachrichten platzieren, die dann von den zuständigen anderen Agenten verarbeitet werden.

4.2.2 Komponentenintegration mit WARP

Für die Integration von Komponenten sind die Agenten der Konfigurationsschicht zuständig. Die Site Manager Agenten (SMA) stellen Informationen über die verfügbaren Dienste (Services) zur Verfügung. Bei WARP erfolgt die Analyse der Services dabei nicht durch Untersuchung des Quellcodes, sondern es werden Introspektionsmechanismen benutzt, um an Informationen über die Zugriffsmöglichkeiten und Schnittstellen der Komponenten zu gelangen (siehe [Bla00] S.1). Die so gewonnenen Daten werden in der gemeinsamen Datenbank abgelegt. Für die verfügbaren Services erzeugt der SMA anschließend Instanzen des Role Manager Agenten (RMA), die die Funktionalität der ihm zugeordneten Dienste in die Workflows einbringen kann. Nachdem bekannt ist, welche Dienste zur Verfügung stehen, generiert der Global Workflow Manager Agent (GWMA) für jeden Service eine grafische Repräsentation. Ein Prozessdesigner kann nun mit Hilfe des Design-Werkzeugs Workflow-Modelle daraus entwerfen.

4.2.3 Workflow-Modellierung mit WARP

Zur Spezifikation eines mit WARP auszuführenden Prozesses müssen verschiedene objektorientierte Modelle erstellt werden. Dazu steht in WARP ein Modellierungswerkzeug zur Verfügung, das jedoch keine Unterstützung bei der Analyse und Simulation von Prozessen bietet. WARP unterscheidet bei den Modellen zwischen struktureller, funktionaler und nicht-funktionaler Sicht.

Im *strukturellen* Modell, das in Form von UML-Klassendiagrammen aufgestellt wird, werden zunächst die vorhandenen Services erfasst. Sie können in einem weiteren Schritt zu Rollen aggregiert werden, um sie dann dem zu modellierenden Workflow zuzuordnen. Die Aggregation zu Rollen ermöglicht es einem Role Manager Agenten, auch mehrere Dienste anzubieten. In der *funktionalen* Modellsicht werden die eigentlichen Abläufe des Workflows beschrieben. Dazu werden in zwei verschiedenen UML-Aktivitätendiagrammen der Daten- und Kontrollfluss des Prozesses getrennt voneinander modelliert. Zu den *nicht-funktionalen* Eigenschaften des Workflows zählen Besonderheiten wie Fehler- und Ausnahmebehandlung, Nebenläufigkeit usw. Leider wird in [Bla00] nicht näher auf die Modellierung der nicht-funktionalen Sicht eingegangen. Die Trennung in funktionale und nicht-funktionale Sicht folgt dabei laut [Bla00] (S.4) dem Prinzip der aspektorientierten Programmierung.

Ein Global Workflow Manager Agent (GWMA) nimmt schließlich diese Spezifikationen des Workflow-Modells entgegen und konvertiert sie in die interne Datenstruktur. Dabei handelt es sich jedoch nicht um ein XML-Format, so wie es in Abschnitt 3.2 gefordert wurde. Anschließend legt er die Modelle in der gemeinsamen Datenbank ab, damit sie von den übrigen Agenten zur Konfiguration und zur Ausführung der Workflow-Modelle benutzt werden können.

4.2.4 Workflow-Ausführung mit WARP

Für das Erzeugen von Workflow-Instanzen und die Kontrolle ihrer Ausführung sind die Agenten der Anwendungsschicht zuständig. Damit die modellierten Workflows ausgeführt werden können, startet der GWMA für jedes Workflow-Modell eine Instanz des Workflow Manager Agent (WMA). Dieser übernimmt die Aufgaben, die sich aus dem nicht-funktionalen Modell des Prozesses ergeben, wie Fehlerbehandlung, Transaktionseigenschaften und ähnliches. Ein WMA übernimmt dabei nicht die vollständige Kontrolle einer Workflow-Instanz sondern ergänzt lediglich die von den nicht-funktionalen Eigenschaften geforderten Aufgaben.

Die Aufgabe der verschiedenen RMA besteht darin, für die Ausführung der Aktivitäten innerhalb der einzelnen Prozessinstanzen zu sorgen. Aufgrund ihres Wissens über die verfügbaren Dienste und ihre Rollen sind sie in der Lage, den jeweils passenden Dienst anzusprechen. Die dazu benötigten Informationen entnehmen sie aus der gemeinsamen Datenbank, in der sich alle notwendigen Modelle und Daten befinden. Um über anfallende Aufgaben während der Ausführung benachrichtigt zu werden bzw. selbst Aufträge an andere Agenten zu verteilen, bedienen sich die Agenten des zentralen Event Servers. Dieses Zusammenspiel der einzelnen Systembestandteile führt zu einer korrekten Abarbeitung der zuvor vom Prozessdesigner definierten Workflow-Modelle.

4.3 E-Business-Plattform sunShine

Bei *sunShine* handelt es sich laut [Sun01] um eine Integrationsplattform für B2C- und B2B-Anwendungen. Mit seiner Hilfe lässt sich sowohl das Layout als auch die Geschäftslogik von Webauftritten realisieren. Im Folgenden wird zunächst sunShine allgemein vorgestellt und daran anschließend detaillierter auf die Systemkomponenten eingegangen, die für die Modellierung und Ausführung von Geschäftsprozessen relevant sind.

Das wichtigste Ziel von sunShine ist die Realisierung von Portalen. Nach [Gso01] ist ein Portal „eine Website, die ihre Besucher beim individuellen Aussuchen von Informationen und Produkten unterstützt, indem es eine Vielzahl von personalisierbaren Informationen und Dienstleistungen rund um das spezielle Thema bietet, die nicht nur aus dem Unternehmen bzw. vom Betreiber stammen müssen.“ Wenn sich das Angebot des Portals auf den wirtschaftlichen Bereich bezieht, spricht man auch von „virtuellen Marktplätzen“.

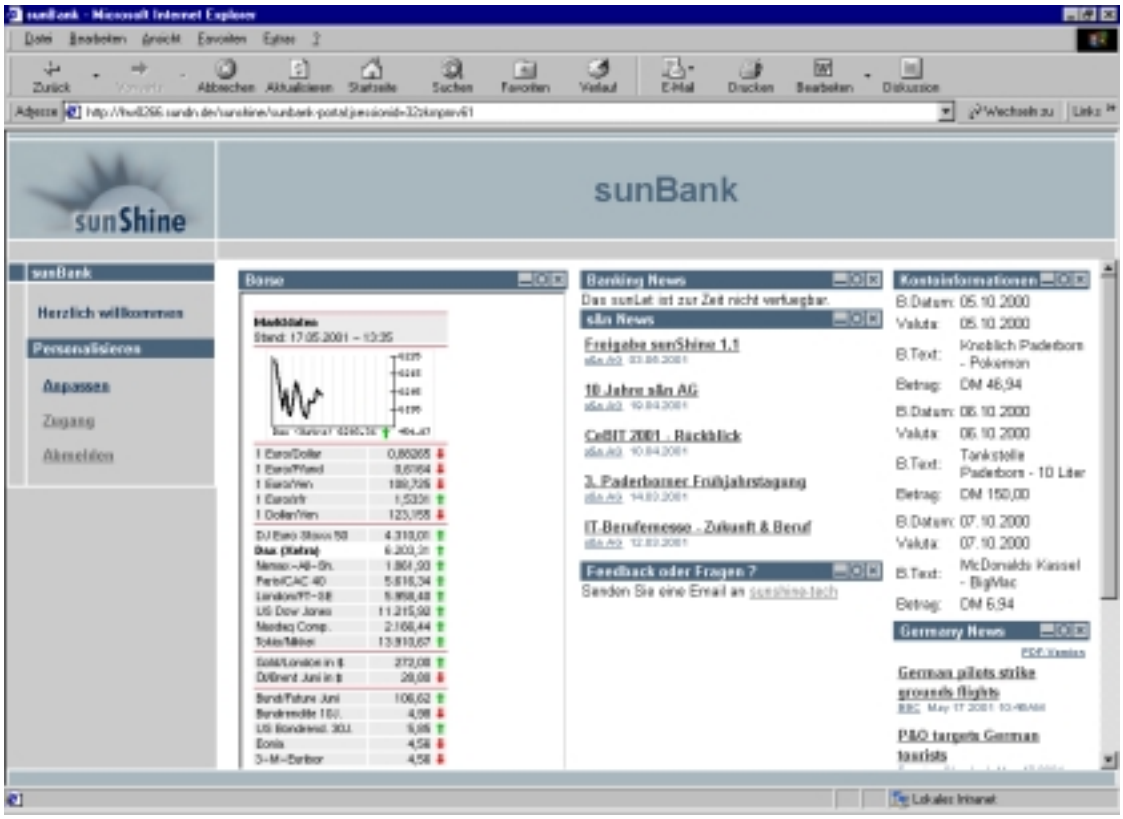


Abb. 4.3: Ansicht eines Beispielportals

Um Informationen, die aus unterschiedlichen Quellen in einem Portal zusammengeführt sind, dem Benutzer individuell anzubieten, sind drei Aspekte zu berücksichtigen: Zunächst sollte ein E-Business-Portal „Multi-Channel-fähig“ sein, d.h. es sollten Daten aus *Quellen unterschiedlichen* Typs eingebunden werden können. sunShine ermöglicht beispielsweise Dateien, Datenbanken, HTTP- oder RMI-Server usw. Um die Daten dem Benutzer zu präsentieren, sollten *verschiedene Ausgabeformate* unterstützt werden. sunShine erfüllt diese Forderung, indem bei einer Anfrage an den Webserver ermittelt wird, mit welchem Endgerät die Anfrage gestellt wurde. Abhängig davon können die

Daten als HTML ausgegeben werden, wenn es sich um einem Browser handelte, als WML, falls die Anfrage von einem WAP-Handy kam o.ä. Schließlich ist es notwendig, dass sich jeder Benutzer mit einem *persönlichen Zugang* am Portal anmeldet. Er hat dann die Möglichkeit, das Aussehen oder die Informationsangebote in bestimmten Grenzen seinen Bedürfnissen entsprechend anzupassen. Diese Konfigurationen werden persistent gespeichert, so dass bei jeder Anmeldung sofort die individuelle Sicht vorhanden ist.

4.3.1 Systemarchitektur von sunShine

sunShine stellt eine Erweiterung des Open-Source-Projekts *Cocoon* der Apache Organisation dar. Ausgangspunkt dieses Projekts war die Beobachtung, dass bei der Realisierung von Web-Anwendungen der Inhalt von Dokumenten, ihr Layout sowie die nötige Geschäftslogik meistens von unterschiedlichen Personengruppen entwickelt werden (siehe [Apa01] und [Lau00]). Bestehende Technologien wie HTML vermischen jedoch häufig diese drei Aspekte, indem beispielsweise Anweisungen, die zur Darstellung eines Dokuments dienen, direkt im Inhalt notiert werden (siehe [HTML99]). Dies führt zu Problemen sowohl bei der Lesbarkeit und Wartbarkeit als auch der Wiederverwendbarkeit der betroffenen Komponenten. Das Ziel von Cocoon ist es daher, die drei Bereiche Inhalt, Layout und Logik strikt voneinander zu trennen. Als plattformunabhängige Technologien für die Umsetzung dieses Ziels werden XML, XSL (eXtensible Stylesheet Language, siehe [XSLT99]) und Java verwendet. In [Apa01] werden folgende drei Ebenen unterschieden:

Erzeugung von XML: Die Erzeugung von XML-Dokumenten geschieht durch Autoren. Sie benötigen kein Wissen über die spätere Verarbeitung ihrer Dokumente, sondern müssen lediglich dafür sorgen, dass diese gültig bezüglich der geforderten Document Type Definition (DTD) bzw. eines XML-Schemas ist.

Verarbeitung von XML: Die erzeugten XML-Dokumente werden schrittweise verarbeitet, indem sie eine Kette von externen Logikkomponenten durchlaufen. Diese können sowohl Daten aus dem Dokument lesen als auch Änderungen daran vornehmen.

Darstellung durch XSL: Die verarbeiteten XML-Dokumente können schließlich durch Anwendung von XSL-Stylesheets in das gewünschte Ausgabeformat transformiert werden. Denkbar sind hier HTML, WML, XML, PDF usw.

Durch die konzeptionelle und implementierungstechnische Unterscheidung dieser drei Ebenen kann eine bessere Trennung von Kompetenzen sowie Wartbarkeit von Softwaresystemen erreicht werden. Cocoon stellt sich dabei als Plug-in-Erweiterung von beliebigen Java-fähigen Webservern dar, um seine Funktionalität zur Verfügung zu stellen. Die Architektur von sunShine übernimmt diese Trennung der drei Ebenen und ergänzt sie um die notwendige Funktionalität zur Erstellung von Portalen. sunShine enthält darüber hinaus mehrere Tools, die bei der Erstellung und Verwaltung von Portalen unterstützen, wie beispielsweise ein Content-Management-System zur Verwaltung der Inhalte des Portals.

4.3.2 Workflow-Ausführung mit sunShine

Hinter jedem Portal können sich Geschäftsprozesse verbergen, die die anfallenden Daten verarbeiten. Einen solchen Geschäftsprozess stellt beispielsweise die durchzuführende Verarbeitung dar, wenn ein Kunde eines Online-Banking-Systems eine Kontotransaktion veranlasst. sunShine bietet durch das Konzept der sogenannten *Pipelines* die Möglichkeit, einfache Prozesse zu beschreiben (vgl. [Lau00]). Da sunShine vollständig XML-basiert ist, wird vorausgesetzt, dass die zu verarbeitenden Daten in Form von XML-Dokumenten vorliegen. Als Black-Box betrachtet, erhält eine Pipeline ein XML-Dokument als Eingabe, führt abhängig von seinem Inhalt Verarbeitungsaktionen durch und gibt schließlich ein manipuliertes XML-Dokument aus.

Abb. 4.4: Pipeline-Konzept

Jeder Pipeline ist eine URL zugeordnet, durch deren Angabe sie beim Webserver angesprochen werden kann. Man spricht daher im Umfeld von Cocoon auch von einer *Ressource*. Cocoon kennt eine Konfigurationsdatei, die sogenannte *Sitemap*, in der alle Ressourcen, die dem Webserver bekannt sein sollen, in einer speziellen XML-Sprache aufgelistet sind. Bislang enthält sunShine kein Werkzeug, mit dem sich solche, durch Pipelines realisierte Workflow-Beschreibungen auf abstrakter Ebene grafisch modellieren lassen. Ebenso wenig ist eine Unterstützung bei der Analyse und Optimierung von Prozessen vorgesehen.

Eine Pipeline besteht aus einem Generator, beliebig vielen Transformern und endet mit einem Serializer. Ein *Generator* stellt die Verbindung zur Datenquelle dar, aus der das XML-Dokument stammt. Er liefert als Ergebnis die gelesenen XML-Daten, die dann zur Verarbeitung an den ersten Transformer weitergeleitet werden. Wie bereits weiter oben erwähnt, kennt sunShine Generatoren für Datenbanken, Dateien, HTTP-Server usw. Die Verarbeitung der XML-Daten durch *Transformer* ist die wichtigste Aufgabe einer Pipeline, daher ist diesem Bereich ein eigener Abschnitt gewidmet (siehe Abschnitt 4.3.3). Wenn die Verarbeitung der XML-Daten abgeschlossen ist, werden sie an einen *Serializer* weitergeleitet. Dieser formatiert das XML-Dokument, indem beispielsweise überflüssige Leerzeilen entfernt werden, und übergibt es dem Webserver, der es als Ergebnis der Anfrage an das Endgerät sendet. Abbildung 4.5 zeigt beispielhaft die Ausführung einer Pipeline:

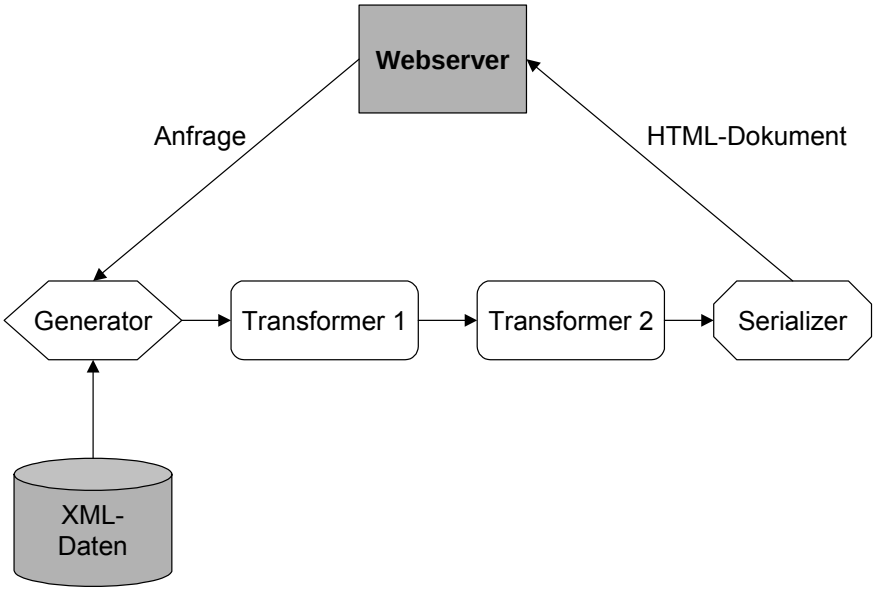


Abb. 4.5: Beispiel der Ausführung einer Pipeline

Im Zusammenhang mit der Ausführung von Pipelines bietet sunShine Ansätze einer Administratorschnittstelle. Sie enthält einfache Funktionen wie die Überwachung von Prozessinstanzen durch Logging-Mechanismen oder ihre Erzeugung über eine API-Schnittstelle.

4.3.3 Komponentenintegration mit sunShine

Die Integration von Komponenten und existierenden Back-End-Systemen geschieht durch die Transformer innerhalb von Pipelines, die die XML-Daten verarbeiten. Jeder Transformer ist dabei eine Java-Klasse, deren Schnittstelle nach außen hin aus den Operationen des SAX-Ereignismodells zur Verarbeitung von XML-Daten besteht (siehe [Meg00]). Ein Transformer erhält nun der Reihe nach alle SAX-Ereignisse des zu verarbeitenden Dokuments. Als Java-Klasse kann er in Abhängigkeit davon alle beliebigen Aktionen ausführen, die in Java möglich sind. Insbesondere lassen sich vorhandene Systemkomponenten ansprechen. Der Transformer kann die Ereignisse so an den jeweils nächsten Transformer der Pipeline weitergeben, wie er sie erhalten hat. Dadurch würden die XML-Daten von ihm nicht verändert. Ebenso kann er aber auch andere SAX-Ereignisse an seinen Nachfolger senden, wenn die Daten manipuliert werden sollen. So ist es beispielsweise denkbar, dass bestimmte Daten des Dokuments herausgefiltert oder neue hinzugefügt werden, die der Transformer aus irgendeiner externen Quelle abgerufen hat. Es ist ihm prinzipiell möglich, das XML-Dokument in beliebiger Weise zu transformieren. Abbildung 4.6 zeigt das allgemeine Modell eines Transformers:

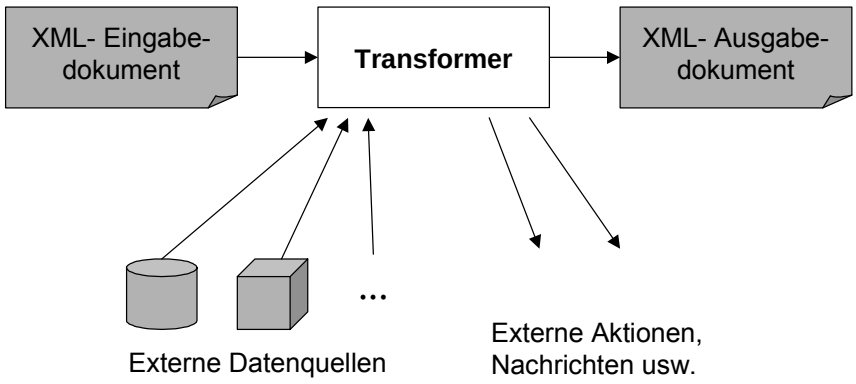


Abb. 4.6: Modell eines Transformers

sunShine bietet eine Reihe von vorgefertigten Transformern an, die als Grundbausteine für die Festlegung eigener Pipelines verwendet werden können. Dabei handelt es sich um Komponenten zum Senden und Empfangen von E-Mails, zum Einbinden von Daten anderer Server oder Datenbanken, insbesondere zur Anwendung von XSL-Stylesheets auf die XML-Daten usw. Es ist aber auch möglich, bei der Entwicklung von Web-Anwendungen eigene Transformer-Komponenten zu entwickeln, die dann in sunShine integriert werden. Durch die einheitliche SAX-Schnittstelle aller Transformer ist es möglich, sie sequentiell hintereinander zu schalten. Sogenannte *Selektoren* erlauben auch einfache Verzweigungen innerhalb solcher Pipeline-Prozesse anhand bestimmter Bedingungen, wie beispielsweise der Art des anfragenden Endgeräts oder der aktuellen Uhrzeit. Durch das Konzept der Pipelines lassen sich also mit sunShine bereits ansatzweise Workflows realisieren. Ein flexibles Workflow-Modell sowie die Unterstützung von Transaktionen sind bislang jedoch nicht vorhanden.

4.4 Bewertung

Nachdem exemplarisch die drei Systeme vorgestellt wurden, soll nun eine Bewertung erfolgen. Dabei wird überprüft, inwieweit sie den im Abschnitt 3.2 aufgestellten Anforderungen gerecht werden. In Tabelle 4.1 sind alle Anforderungen und die Bewertung der einzelnen Systeme in Kurzform aufgelistet.

Anforderung	**BizTalk**	**WARP**	**sunShine**
Modellierungswerkzeug	+	+	-
Analyse, Optimierung und Konsistenzprüfung	-	-	-
Anbindung von Back-End-Systemen	+	+	+
Workflow-relevante Daten	2)	2)	+
Verarbeitung von XML-Dokumenten	+	-	+
Workflow-Beschreibungen in XML	+	-	+
Zuord. von Workflow-Elementen auf reale Komp.	+	3)	3)
Kontrolle und Ausführung der Prozessinstanzen	+	+	+
Portalfunktionalität	-	-	+
Multi-Channel-Fähigkeit	-	-	+
Benutzerinteraktion	2)	2)	2)
Administrator-Schnittstelle	+	1)	o
Flexibles Workflow-Modell für Geschäftsregeln	+	+	-
Transaktionsunterstützung	+	+	-
Plattformunabhängigkeit	-	+	+

1) Keine positive Information auffindbar, also vermutlich nicht vorhanden.
2) Behandlung nicht implizit vorgesehen, aber mit vorhandenen Mitteln realisierbar.
3) Im Modell können nur bereits vorhandene Komponenten verwendet werden.

+ = vorhanden - = nicht vorhanden o = ansatzweise vorhanden

Tab. 4.1: Bewertung vorhandener Workflow-Systeme

Das Ziel von BizTalk ist die Integration von bestehenden, heterogenen Systemen. Damit erfüllt es die Anforderung, dass durch Einführung der neuen E-Business-Plattform nicht ein vollständig neues System entwickelt werden soll, sondern die bestehenden Back-End-Systeme eingebunden und weiterhin verwendet werden. Der ausdrückliche Wunsch nach dem Einsatz von XML als internes Datenformat stellt kein Problem dar, da BizTalk selbst in großen Teilen auf XML-Technologien setzt. Ein Workflow-Management-System, das die Ausführung und Kontrolle der Prozesse übernimmt, steht zur Verfügung.

Ein Mangel von BizTalk besteht darin, dass es keine integrierte Portalfunktionalität besitzt. Da jedoch beispielsweise in der Fallstudie ein wesentliches Element der E-Business-Plattform ein Web-Auftritt mit Multi-Channel-Fähigkeit ist, müsste zu diesem Zweck auf ein separates Produkt zurückgegriffen werden, das in das BizTalk-System eingegliedert wird. Im Gegensatz zur Fallstudie geht es bei BizTalk eher um langlebige Geschäftsprozesse als um systemorientierte Informationsprozesse im Sinne von Ab-

schnitt 2.2.2. Weiterhin kann die Forderung nach Plattformunabhängigkeit nicht erfüllt werden, da das BizTalk-System stark von Microsoft-Produkten abhängig ist.

Demgegenüber fokussiert WARP stärker diesen Bereich des Workflow-Management, wobei das vorrangige Ziel die Konfiguration und Integration heterogener Komponenten ist. Es wäre daher kein Problem, bestehende Back-End-Systeme zu integrieren. Nachteilig wirkt sich allerdings aus, dass die Modellierung der Prozesse durch das Aufstellen der großen Anzahl verschiedener Diagramme relativ aufwendig erscheint und außerdem nur Komponenten eingesetzt werden können, deren Schnittstelle durch Introspektion analysiert werden kann. Wie schon bei BizTalk ist es mit Hilfe von WARP nicht möglich, ein Web-Portal zu erstellen, ohne auf zusätzliche Produkte zurückzugreifen. Besonders nachteilig wirkt sich aus, dass WARP keinerlei XML-Unterstützung anbietet, da dies eine wichtige Anforderung im Bereich E-Business ist.

sunShine stellt ebenfalls einen interessanten Ansatzpunkt für die Integration von Softwarekomponenten dar. Es erlaubt das Einbinden verschiedenster Systeme und Datenquellen in die Verarbeitungsabläufe eines Web-Portals. Daher stellt es kein Problem dar, die Back-End-Systeme aus der Fallstudie zu integrieren und zudem einen Web-Auftritt zu realisieren. Die Forderung nach Multi-Channel-Fähigkeit, d.h. der Möglichkeit, verschiedene Ein- und Ausgabegeräte zu unterstützen, kann ebenfalls erfüllt werden. Alle Werkzeuge, die für die Verarbeitung von XML-Daten benötigt werden, stehen bei sunShine implizit zur Verfügung.

Für die Realisierung von komplexen Workflow-Modellen hat sunShine jedoch zwei Schwächen: Zum einen ist jede Pipeline, d.h. die Abfolge ihrer Bestandteile wie Transformer, Selektoren usw. fest in der Cocoon-Sitemap durch eine spezielle XML-Sprache codiert. Dies entspricht einer Programmierung der Workflows und steht im Widerspruch zur Forderung nach einer visuellen Modellierung von Prozessabläufen. Zum anderen müsste eine Erweiterung des Pipeline-Konzepts stattfinden, um Transformer flexibler als bisher verketten und somit komplexe Workflow-Modelle, wie beispielsweise die Klassifizierung der Dokumenttypen in der Fallstudie, aufbauen zu können.

Die Betrachtung der ausgewählten Systeme, die den augenblicklichen Stand der Technik widerspiegelt, hat gezeigt, dass bisher noch keine umfassende Lösung für die sich unter anderem aus der Fallstudie ergebenen Anforderungen an eine prozessorientierte Integration von Softwarekomponenten im Bereich des E-Business existiert. Trotzdem ist deutlich geworden, dass der Einsatz von Workflow-Modellen und deren rechnerunterstützte, automatisierte Ausführung einen guten Ansatz darstellt, um die Integration und Anbindung bestehender Software-Anwendungen eines Unternehmens an elektronisch abzuwickelnde Geschäftsvorfälle und Prozesse voranzutreiben. Im weiteren Verlauf dieser Arbeit soll daher sunShine um eine Workflow-Management-Komponente erweitert werden, die die flexible Modellierung und Ausführung von Geschäftsprozessen in realen E-Business-Systemen ermöglicht (siehe Kapitel 9 und 10).

5 Evaluierung vorhandener Workflow-Modellierungssprachen

Im vorangegangenen Kapitel wurden drei Workflow-Systeme zur Softwareintegration im E-Business betrachtet. Da im Abschnitt 3.2 die Forderung nach einer visuellen Modellierung von Workflows aufgestellt wurde, sollen nun grafische Sprachen untersucht werden, mit denen sich Workflow-Modelle entwerfen und beschreiben lassen. Als bekannteste Vertreter wurden Petri-Netze, Ereignisgesteuerte Prozessketten (EPK) und UML-Aktivitätendiagramme ausgewählt.

5.1 Petri-Netze

Petri-Netze sind ein bekanntes Mittel zur Beschreibung und Analyse von Prozessabläufen. Daher soll in diesem Abschnitt untersucht werden, ob sie sich als Modellierungssprache für XML-Prozesse eignen. Zunächst erfolgt eine allgemeine Einführung in die Theorie der Petri-Netze und ihre Anwendbarkeit für allgemeine Geschäftsprozesse. Anschließend werden die Möglichkeiten von Petri-Netzen mit den Anforderungen von XML-Prozessen verglichen.

5.1.1 Klassische Petri-Netze

Ein klassisches Petri-Netz ist ein gerichteter, bipartiter Graph mit zwei Knotenmengen, *Stellen* und *Transitionen*. In der grafischen Darstellung von Petri-Netzen werden Stellen durch Kreise und Transitionen durch Rechtecke repräsentiert. Ein solches Netz lässt sich durch folgende formale Definition beschreiben (vgl. [Bau96] S.50 und [Aal97]):

> **Definition:** Ein Petri-Netz ist ein 3-Tupel (S, T, F).
> Dabei ist S eine endliche Menge von Stellen
> und T eine endliche Menge von Transitionen mit $(S \cap T) = \emptyset$,
> sowie $F \subseteq (S \times T) \cup (T \times S)$ die Menge der Kanten zwischen S und T.

Eine Stelle s heißt *Eingangsstelle* einer Transition t, wenn es eine gerichtete Kante von s nach t gibt. Analog dazu handelt es sich um eine *Ausgangsstelle*, wenn eine Kante in der umgekehrten Richtung, also von t nach s existiert.

Um die Dynamik von Petri-Netzen zu beschreiben, werden sogenannte *Marken* verwendet. Jede Stelle eines Netzes enthält zu jeder Zeit keine, eine oder mehrere Marken. Diese werden grafisch durch schwarze Punkte innerhalb der Stelle dargestellt. Eine Transition gilt dann als *aktiviert*, wenn jede ihrer Eingangsstellen mindestens eine Marke enthält. Jede aktivierte Transition kann *schalten*. In diesem Fall wird aus jeder ihrer Eingangsstellen genau eine Marke entfernt und in jeder Ausgangsstelle je eine neue Marke erzeugt. Die folgende Abbildung zeigt ein einfaches Petri-Netz vor und nach dem Schaltvorgang der Transition t2.

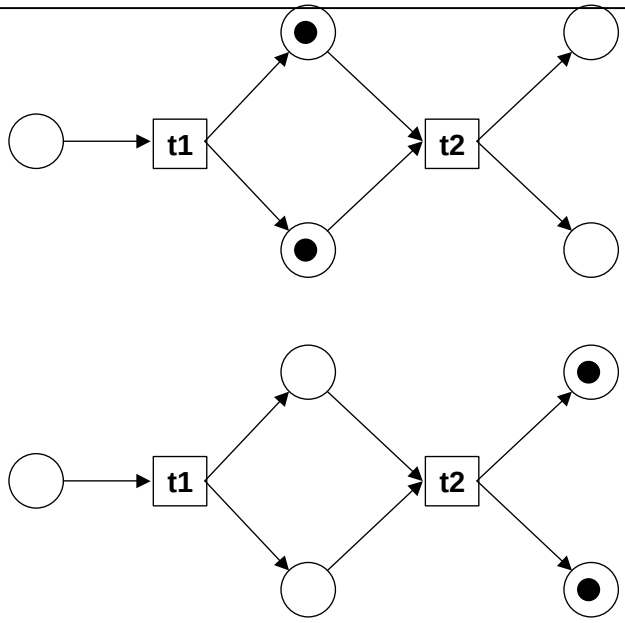


Abb. 5.1: Petri-Netz vor und nach einem Schaltvorgang

Mit Hilfe dieser klassischen Form von Petri-Netzen lassen sich zwar bereits einfache Prozesse modellieren, es fehlen jedoch mächtigere Sprachkonstrukte, um die Semantik eines Modells verfeinern zu können. Aus diesem Grund gibt es viele Ansätze zur Erweiterung von Petri-Netzen, die je nach Einsatzgebiet zusätzliche Sprachelemente definieren. Solche speziellen Varianten von Petri-Netzen werden unter dem Begriff *High-Level-Petri-Netze* zusammengefasst. Im folgenden Abschnitt werden einige Erweiterungen vorgestellt, die besonders für den Bereich der Geschäftsprozessmodellierung von Bedeutung sind.

5.1.2 High-Level-Petri-Netze

Im Zusammenhang mit der Modellierung von Geschäftsprozessen wurden vor allem drei wichtige Erweiterungen der klassischen Petri-Netze vorgeschlagen, um mächtigere Ausdrucksmöglichkeiten zur Verfügung zu haben. Dabei handelt es sich um individuelle Marken, Zeit und Hierarchie.

Eine gravierende Schwäche herkömmlicher Petri-Netze ist die Tatsache, dass alle Marken gleichwertig und damit nicht unterscheidbar sind. Beim realen Einsatz von Petri-Netzen zur Modellierung von Prozessen stellen Marken jedoch häufig Objekte dar, die am Prozess beteiligt sind. Dies können Menschen, Ressourcen, Dokumente o.ä. sein. Es ist nun wünschenswert, Aussagen über bestimmte Eigenschaften solcher Objekte zu machen, sei es als Vorbedingung zum Schalten einer Transition oder als Spezifizierung ihres Ergebnisses. Um dies modellieren zu können, stellt [Bau96] (S.195) folgenden Ansatz vor: Jede Marke in einem Petri-Netz ist ein wohldefiniertes mathematisches Objekt, also beispielsweise eine Zahl, Zeichenkette, ein Tupel usw. Auf diese Weise werden Marken unterscheidbar, d.h. *individuell*, und es lässt sich sogar mit ihnen rechnen. Für jede Transition lassen sich nun Aussagen über die Marken in ihren Eingangs- und Ausgangsstellen machen. Abbildung 5.2 zeigt ein einfaches Beispiel.

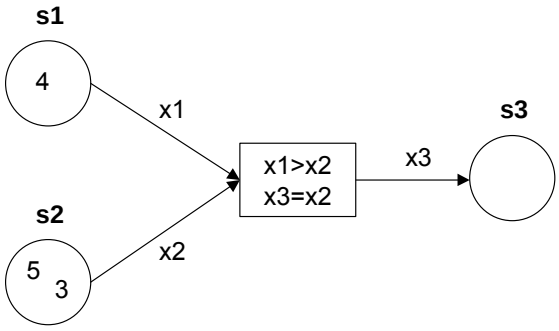


Abb. 5.2: Petri-Netz mit individuellen Marken

Bei einer solchen Konstellation, wie in der Abbildung dargestellt, könnte nur die Marke „3“ verarbeitet werden, da die Transition fordert, dass Marke x2 aus Stelle s2 kleiner sein muss als die Marke x1 aus der Stelle s1, in diesem Fall „4“. Für die Marke „5“ trifft diese Bedingung nicht zu. Wenn die Transition schaltet, würden die Marken „3“ und „5“ aus ihren Stellen entfernt und stattdessen eine Marke mit dem Wert „3“ in der Stelle s3 erzeugt, da die Transition festlegt, dass die Marke x3 den Wert von x2 erhält.

Eine zweite wichtige Erweiterung stellt die Modellierung von *zeitlichem Verhalten* dar. Klassische Petri-Netze gehen davon aus, dass eine Transition schaltet, sobald sie aktiviert ist und dass die Ausführung des Schaltvorgangs keine Zeit in Anspruch nimmt. Solche Bedingungen sind in der Praxis so gut wie nie erfüllt. Vielmehr muss man spezifizieren können, dass Marken sich eine bestimmte Zeit lang in einer Stelle aufhalten, bevor sie diese verlassen, oder dass eine Aktion, also das Schalten einer Transition, eine gewisse Zeit dauert. Auch Aussagen wie „Wenn eine Marke sich eine bestimmte Zeit lang in einer Stelle befindet, ohne von den zuständigen Transitionen bearbeitet worden zu sein, schaltet eine spezielle Transition und lenkt die Marke auf einen anderen Weg“ sind denkbar und müssen modellierbar sein. Da explizite Aussagen über zeitliche Aspekte in unserem Bereich der Prozessmodellierung jedoch nicht von zentraler Bedeutung sind (vgl. Abschnitt 3.4), wird an dieser Stelle auf eine detaillierte Einführung geeigneter Sprachkonstrukte verzichtet. Für eine tiefergehende Darstellung von Zeitmodellierung sei auf [Ajm95] über stochastische Petri-Netze verwiesen.

Bei der präzisen Spezifikation realer Systeme werden Petri-Netze häufig groß, komplex und damit unübersichtlich. Daher ist es überaus sinnvoll, eine Art *Hierarchie* bzw. Schachtelung von Netzen zu erlauben. Hinter einer einzelnen Transition oder auch Stelle kann sich dann ein Subnetz verbergen. Dabei handelt es sich um ein eigenständiges Petri-Netz, das eine Teilaufgabe beschreibt, die ausgeführt wird, wenn sich Marken in der Stelle des übergeordneten Netzes befinden bzw. die Transition schaltet. Diese Möglichkeit der Schachtelung hilft nicht nur dabei, Netze übersichtlich zu halten, sondern erlaubt auch die logische Gliederung eines Prozesses in Teilprozesse.

5.1.3 Geschäftsprozessmodellierung mit Petri-Netzen

Das Ziel von Workflow-Management-Systemen ist die Definition, Kontrolle und Ausführung von Arbeitsabläufen. Da Petri-Netze ein bekanntes Mittel zur Beschreibung von Prozessen sind, gibt es eine Reihe von Gründen, Petri-Netze zur Modellierung von Geschäftsprozessen einzusetzen (vgl. [Aal97]):

Grafische Sprache: Als grafische Sprache sind Petri-Netze recht intuitiv und einfach zu erlernen. Zudem lassen sich grafische Darstellungen im Allgemeinen besser verstehen und bieten eine geeignete Grundlage zur Diskussion nicht nur zwischen Spezialisten sondern in gewissem Maße auch mit Laien.

Formale Semantik: Petri-Netze und ihre zahlreichen Erweiterungen besitzen eine mathematisch exakt definierte formale Semantik. Diese ermöglicht nicht nur die präzise Beschreibung der zu modellierenden Sachverhalte, sondern ist auch die Grundvoraussetzung, um solche Modelle automatisiert interpretieren und ausführen zu können.

Ausdrucksstärke: Besonders durch die Vielzahl an Erweiterungen gegenüber den klassischen Netzen verfügen Petri-Netze über nahezu alle erforderlichen Sprachkonstrukte, die für die Beschreibung komplexer Systeme und Prozesse notwendig sind.

Analysemöglichkeiten: Aus der mathematischen Basis der Theorie von Petri-Netzen ergeben sich vielfältige Analysemöglichkeiten konkreter Netze. Es lassen sich Aussagen über Erreichbarkeit, Antwortzeiten, Deadlocks und andere Eigenschaften beweisen. Auf diese Weise hat man mächtige Mittel zur Hand, um Netze verifizieren oder optimieren zu können.

Herstellerunabhängigkeit: Schließlich ergibt sich ein Vorteil daraus, dass Petri-Netze nicht von einem bestimmten Hersteller abhängen. Sie bilden einen unabhängigen Rahmen für die Modellierung und Analyse von Prozessen, der nicht durch wirtschaftliche Interessen und Entscheidungen beeinflusst wird.

Für den Einsatz von Petri-Netzen im Bereich der Workflow-Modellierung schlägt [Aal97] folgende Abbildung von Geschäftsprozessen auf Petri-Netze vor. Da jeder Geschäftsprozess laut Definition eine Folge von Aktionen mit gemeinsamem Geschäftsziel ist [Wmc95], muss zunächst genau eine definierte Startstelle (Quelle) und genau eine Zielstelle (Senke) existieren. Jede weitere Stelle repräsentiert einen bestimmten Zustand innerhalb der Prozessablaufs. Aufgaben, die während des Prozesses durchzuführen sind, werden durch Transitionen ausgedrückt. Jeder Anwendungsfall, also beispielsweise jedes Dokument, das einen Prozess durchläuft, wird durch Marken dargestellt. Mit Hilfe der Erweiterung von individuellen Marken ist es möglich, die einzelnen Anwendungsfälle zu unterscheiden und zusätzliche Anwendungsdaten daran zu knüpfen, die für die Ausführung des Prozesses relevant sind.

Abbildung 5.3 zeigt zur Verdeutlichung das Petri-Netz eines beispielhaften Geschäftsprozesses. Modelliert wurde die Situation, dass ein Bankkunde bei seiner Bank einen Kredit beantragt. Je nach Höhe des Kredits bzw. dem Ergebnis einer Schufa-Anfrage verzweigt die Ausführung in unterschiedliche Äste. Um einen besseren Vergleich der verschiedenen Modellierungssprachen zu ermöglichen, findet sich dieses Beispiel auch in den Abschnitten 5.2 und 5.3 wieder.

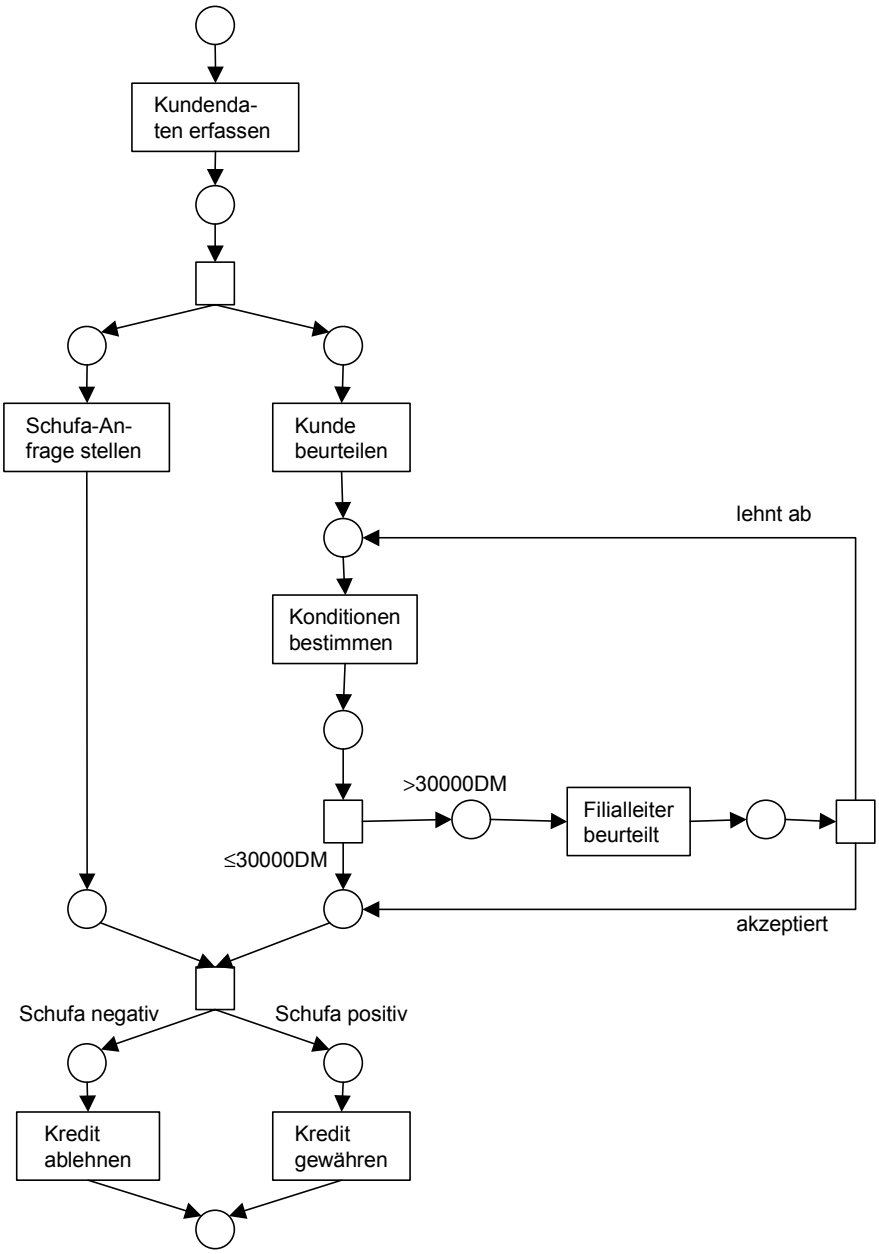


Abb. 5.3: Beispiel eines Petri-Netzes

5.1.4 Eignung für Prozessdiagramme

Im Abschnitt 3.3 wurden Anforderungen an die Workflow-Modellierungssprachen im Anwendungsgebiet der Integration von Softwarekomponenten mit Hilfe von XML-Technologie formuliert, die von einer geeigneten Sprache erfüllt werden müssen. Daher soll nun der Reihe nach untersucht werden, inwieweit die Theorie der Petri-Netze die einzelnen Forderungen erfüllt und damit für die Modellierung verwendet werden kann.

Als visuelle Sprache zur Beschreibung von Prozessabläufen kommen Petri-Netze grundsätzlich in Frage. Sie erlauben eine anschauliche und leicht zu verstehende Darstellung. Im Zusammenhang mit Geschäftsabläufen treten vier verschiedene Basiskonstrukte bezüglich des Kontrollflusses auf: sequentielle Abarbeitung von Aktionen, bedingte Verzweigung des Prozesses, iterative Ausführung von Aktionsfolgen sowie

parallele Verarbeitungsstränge. Alle vier Möglichkeiten können in Petri-Netzen problemlos ausgedrückt werden. Nach [Aal97] existieren dazu die Grundbausteine, die in der folgenden Abbildung dargestellt sind.

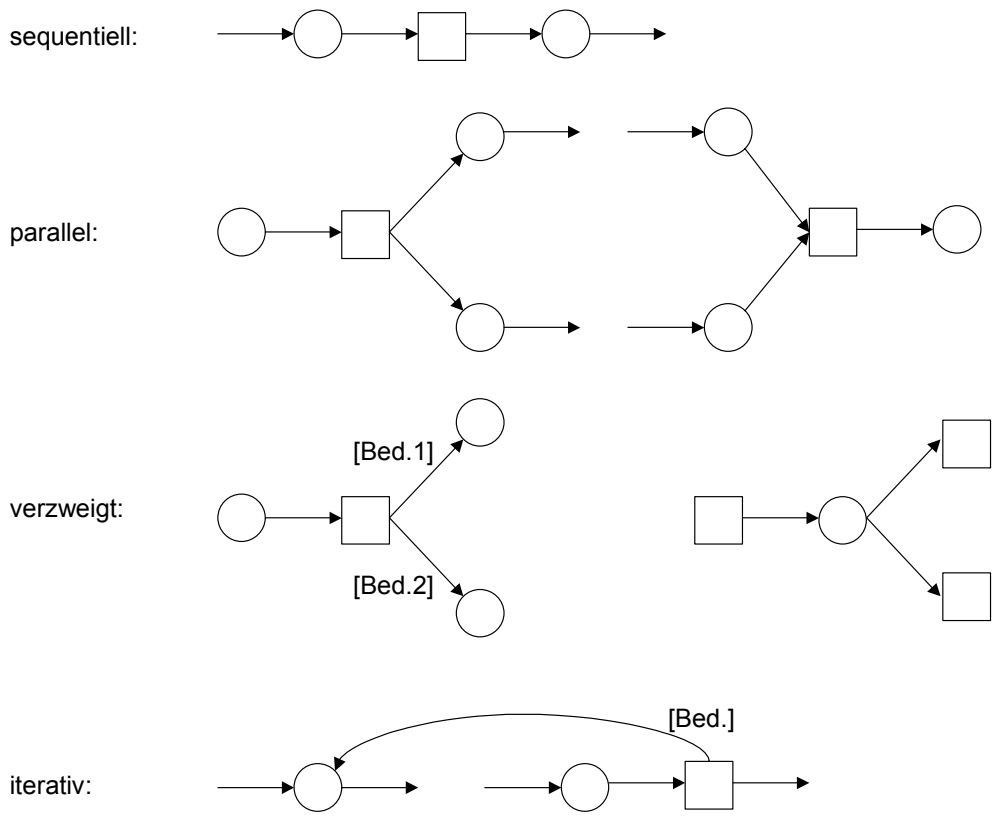


Abb. 5.4: Kontrollflusskonstrukte in Petri-Netzen

Sequentielle Aktionsfolgen werden durch einfache Hintereinanderschaltung von Stellen und Transitionen erreicht. Um Parallelität zu erreichen, erzeugt eine Transition mehrere Marken in ihren Ausgangsstellen. Dieser Vorgang wird auch *AND-Split* genannt. Jede Marke läuft nun parallel durch einen eigenen Aktionsstrang, um schließlich durch einen *AND-Join* wieder mit den anderen zusammengeführt zu werden. Auf diese Weise findet eine Synchronisation der parallelen Teilprozesse statt, da die Transition erst dann schalten kann, wenn alle Marken in den Eingangsstellen der Transition angekommen sind. Es ist auch möglich, verschiedene parallele Ausführungsstränge an beliebigen Stellen zu synchronisieren. Das Basiskonstrukt dazu ist in Abbildung 5.5 dargestellt.

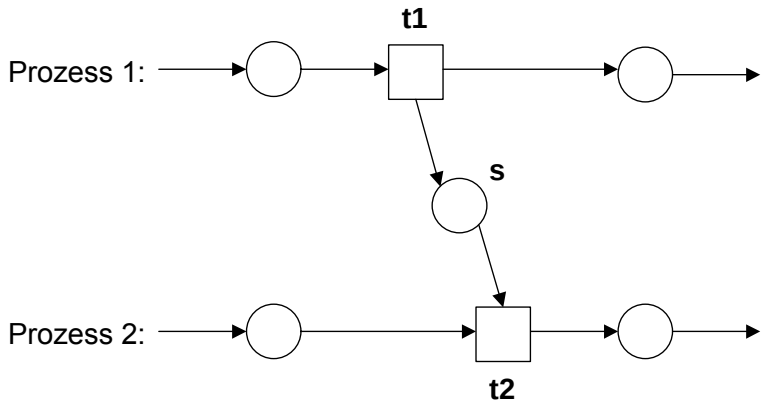


Abb. 5.5: Synchronisation von parallelen Teilprozessen

Wenn Prozess 1 am Synchronisationspunkt angekommen ist, erzeugt seine Transition t1 eine Marke in der Stelle s. Damit wird dem Partnerprozess die Ankunft signalisiert. Wenn Prozess 2 am Synchronisationspunkt angekommen ist, kann er erst dann weiterlaufen, nachdem die Marke vom Partnerprozess erzeugt wurde. Das Schalten der Transition t2 löscht die Marke und damit das Signal, damit bei der nächsten Ausführung die Synchronisation korrekt stattfindet. In diesem Beispiel wartet also Prozess 2 auf Prozess 1. Um eine gegenseitige Synchronisation durchzuführen, muss das Basiskonstrukt zweifach verwendet und gespiegelt in das Netz eingebaut werden. Analog lässt sich auch die Synchronisation von mehr als zwei Prozessen beschreiben.

Bei der bedingten Verzweigung unterscheidet man zwei Fälle. Beim *expliziten OR-Split* werden an den Kanten zwischen einer Transition und ihren Ausgangsstellen Bedingungen notiert. Nur die Ausgangsstelle, deren Bedingung erfüllt ist, wird beim Schalten markiert. Auf diese Weise findet die weitere Verarbeitung nicht in parallelen Teilsträngen statt wie beim AND-Split, sondern genau ein Strang wird gewählt, in dem der Prozess fortfährt. Demgegenüber besitzen beim *impliziten OR-Split* mehrere Transitionen dieselbe Eingangsstelle. Nur eine Transition kann die Marke, die sich in der Stelle befindet, konsumieren. Welche der Transitionen letztlich schalten darf, muss separat modelliert werden. Hier könnten beispielsweise zusätzliche Eingangsstellen für jede Transition eingeführt werden, die die nötigen Bedingungen repräsentieren.

Iterative Prozessabläufe werden durch Verzweigungen realisiert. Abhängig von bestimmten Bedingungen verzweigt der Prozessablauf zurück zu einer früheren Stelle. Die Forderung, dass ein Workflow-Modell immer genau einen Start- und einen Endpunkt haben muss, ist zwar durch Petri-Netze prinzipiell nicht erfüllt. Dies kann aber einfach als zusätzliche Bedingung an ein Petri-Netz gestellt werden. Im Rahmen eines Modellierungswerkzeugs lässt sich diese Eigenschaft eines Netzes leicht überprüfen und sicherstellen. Durch die Erweiterung von Petri-Netzen um Hierarchie, d.h. die Möglichkeit Subnetze zu definieren, die aus einem übergeordneten Netz heraus aufgerufen werden, ist es möglich, vordefinierte Basisprozesse zu einem neuen Geschäftsvorgang zu aggregieren.

In Petri-Netzen wird durch das Schalten von Transitionen der Durchlauf von Objekten durch einen Arbeitsablauf beschrieben. Wie bereits beschrieben lassen sich verschiedene Kontrollflüsse modellieren, wie parallele Ausführung, Synchronisation usw. Es wird jedoch nicht formal spezifiziert, welche Aktivität sich hinter der Ausfüh-

rung einer Transition verbirgt. Während klassische Netze sogar davon ausgehen, dass das Schalten keine Zeit in Anspruch nimmt, kann man mit höheren Netzen Aussagen über solchen Zeitverbrauch machen. Derartige explizite Angaben über die Ausführungsdauer von Prozesskomponenten ist für unseren Anwendungsbereich nicht erforderlich. Stattdessen muss definiert werden, welche Komponente in welchem Zustand des Prozesses angesprochen werden soll, um die Verarbeitung zu übernehmen. Da sich Petri-Netze ausschließlich auf die Modellierung der dynamischen Abläufe beschränken, kann ein statisches Modell der Komponenten eines Prozesses nicht dargestellt werden. Ebenso wenig ist es in Petri-Netzen möglich, Parametereinstellungen für den Aufruf der Komponente anzugeben, der sich hinter dem Schalten einer Transition verbirgt. Um diese Anforderungen erfüllen zu können, müssten Petri-Netze um geeignete Sprachelemente erweitert werden. Diese Erweiterung müsste exakt definiert werden, damit die formale Semantik der Netze erhalten bliebe und eine automatisierte Ausführung möglich wäre. Die Forderung nach einem Konstrukt zur Umsetzung eines Transaktionskonzepts lässt sich mit Petri-Netzen ebenfalls nicht ohne Weiteres realisieren.

Da dem Datenfluss, also der Verarbeitung und Manipulation von Anwendungsdaten während der Ausführung eines Prozesses, in dieser Arbeit eine besondere Rolle zukommt, ist diesem Aspekt ein eigener Abschnitt gewidmet.

5.1.5 Datenfluss in Petri-Netzen

In realen Prozessen gibt es fast immer bestimmte Objekte, die während der Ausführung von Aktionen manipuliert und zwischen verschiedenen Aktionen weitergereicht werden. Solche Sachverhalte, die man auch Daten- oder Objektfluss nennt, müssen in der Modellierung korrekt abgebildet und beschrieben werden. In Petri-Netzen besteht die einzige Ausdrucksmöglichkeit darin, Objekte mit Marken zu identifizieren. Der Ausführungssemantik von Petri-Netzen entsprechend, werden sie beim Schalten der Transitionen von Stelle zu Stelle weitergeleitet. Um verschiedene Objekte, die sich im Netz befinden, unterscheiden zu können, muss man auf die Technik der individuellen Marken zurückgreifen. Auf diese Weise können sowohl Objekte, die zu unterschiedlichen Anwendungsfällen, als auch verschiedene Dokumente innerhalb desselben Anwendungsfalls eindeutig identifiziert werden. Weiterhin ist es möglich, beliebige Attribute und deren Werte an eine Marke zu binden, um so Manipulationen an den Objekten beschreiben zu können.

Bei den in dieser Arbeit betrachteten XML-basierten Workflows erhält jeder Prozess bei seiner Ausführung ein XML-Dokument als Eingabe (siehe Abschnitt 6.1). Dieses kann während des Durchlaufs manipuliert werden und bildet schließlich die Ausgabe bzw. das Ergebnis des Prozesses. Eine geeignete Sprache muss so konstruiert sein, dass lesender und schreibender Zugriff auf dieses Dokument beschrieben werden kann. Einen interessanten Ansatz in dieser Richtung haben Kirsten Lenz et al. in [Len01] mit sogenannten *XML-Netzen* vorgestellt. Das Ziel bei der Entwicklung von XML-Netzen war die Verbindung der Flussmodelle von XML-Dokumenten mit Geschäftsprozessen und deren Ausführung. Ein zentraler Aspekt in XML-Netzen ist die Gültigkeit von XML-Dokumenten bezüglich DTDs. Zur grafischen Beschreibung von DTDs dient dabei die Sprache *Graphical XML Schema Definition Language (GXSL).*

Ein XML-Netz ist laut [Len01] ein Petri-Netz mit den folgenden Erweiterungen: Jeder Stelle ist ein *GXSL-Schema,* also eine DTD zugeordnet, die den Typ der zulässi-

gen Dokumente spezifiziert. Daher können Stellen als Container für XML-Dokumente interpretiert werden, welche gültig bezüglich der zugeordneten DTD sind. Der Fluss von XML-Dokumenten durch das Netz entsteht durch das Schalten von Transitionen. Jede Kante zwischen einer Stelle und einer Transition besitzt ein gegenüber dem Schema der Stelle „erweitertes GXSL-Schema“. Nur XML-Dokumente in der Stelle, die dem erweiterten GXSL-Schema entsprechen, können von der Transition verarbeitet werden. Auf diese Weise lassen sich Verzweigungsbedingungen realisieren. Jeder Kante zwischen einer Transition und einer Stelle ist ein *XManiLa-Ausdruck* zugeordnet. Dabei handelt es sich um eine grafische Sprache zur Spezifikation von Manipulationen an XML-Dokumenten. Beim Schalten der Transition werden die so beschriebenen Änderungen an dem zu verarbeitenden XML-Dokument vorgenommen.

XML-Netze sind für die Modellierung der XML-Prozesse aus einigen Gründen nicht optimal geeignet. In XML-Prozessen soll die Manipulation der Daten nicht im Modell spezifiziert werden. Es wird lediglich angegeben, welche Softwarekomponente die Daten verarbeiten soll. Dabei ist für jede Komponente bekannt, welche Anforderungen sie an ihre Eingabedaten stellt und welche Ausgabedaten sie erzeugt. Was genau intern geschieht, ist dem Modell hingegen nicht bekannt. Diese Annahme ist mit dem für XML-Netze gewählten Ansatz der XManiLa-Ausdrücke nicht vereinbar, da mit XManiLa Manipulationen an XML-Dokumenten beschrieben werden. Schließlich erlauben XML-Prozesse auch Aktionen, die unabhängig von den XML-Daten sind. Denkbar ist beispielsweise das Versenden von E-Mails oder die Ausführung von Datenbankoperationen. Da XML-Netze sich ausschließlich auf die Verarbeitung von XML-Daten konzentrieren, ist eine Spezifizierung solcher Aktionen nicht möglich.

5.2 Ereignisgesteuerte Prozessketten

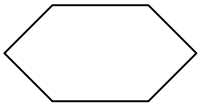
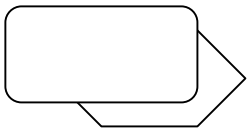
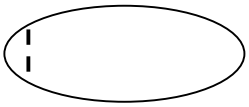
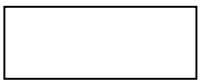
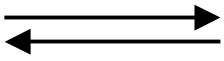
Das Konzept der Geschäftsprozessmodellierung mit „*Ereignisgesteuerten Prozessketten*“ (EPK) wurde Anfang der neunziger Jahre entwickelt und geht auf Arbeiten im Umfeld von Scheer zurück (vgl. [Kel92]). Der in [Kel92] als EPK vorgestellte „prozeßorientierte Modellierungsansatz bildet die Grundlage zur Ermittlung und Dokumentation der betriebswirtschaftlichen Zusammenhänge in einem Unternehmen“ (S.1). Mit ihm sollen „Geschäftsprozesse einfach, übersichtlich und trotzdem eindeutig dargestellt werden“ ([Kel99] S.158). Der Zweck des Einsatzes der EPK ist also in erster Linie die Dokumentation der betrieblichen Zusammenhänge und Abläufe, eine mögliche automatisierte Ausführung wurde zu Beginn nicht berücksichtigt. Es wird sich im Folgenden auch zeigen, dass das Ziel der Eindeutigkeit nur bedingt erreicht werden kann.

5.2.1 Basiskonstrukte des EPK-Modells

Das EPK-Modell ist im Kontext der von Scheer entwickelten „*Architektur integrierter Informationssysteme*“ (ARIS) entstanden. Bei diesem Ansatz wird die Gesamtsicht auf ein Unternehmen in fünf Teilsichten aufgeteilt. Die vier Sichten Funktionssicht, Organisationssicht, Datensicht und Leistungssicht spiegeln die Struktur des Systems wider, während die Steuerungs- bzw. Prozesssicht die dynamischen Verhaltensaspekte von Geschäftsprozessflüssen behandelt (vgl. [Sch98] S.36f). Für die Modellierung der für die Steuerungssicht relevanten Prozesse wird bei ARIS das EPK-Modell benutzt. Dabei dient die EPK in dieser Sicht als eine integrative Darstellung, die neben den dynami-

schen Aspekten auch funktionale und datenorientierte Zusammenhänge aufzeichnet (siehe [Wen00] S.33).

Eine EPK wird als ein Vorgangskettendiagramm dargestellt. Das Grundprinzip ist dabei der „zeitlich-sachlogische Ablauf von Ereignissen und Aufgaben“ ([Kel99] S.158). Bei der Darstellung werden Elemente als Knoten eines Graphen gezeichnet, deren Beziehungen als Kanten in diesem Graph erscheinen. Mit Verknüpfungsoperatoren lässt sich die Ablauflogik des Kontrollflusses abbilden. Die Sprachkonstrukte im Einzelnen sind in der folgenden Tabelle (nach [Kel99] S.161) aufgeführt.

Bezeichnung	Symbol	Definition
Ereignis		Beschreibt einen Zustand, der eine Folge bewirkt: Wann soll etwas gemacht werden?
Funktion		Beschreibt eine Transformation von einem Eingangszustand in einen Zielzustand: Was soll gemacht werden?
Verknüpfungsoperatoren		Beschreibt die logischen Verbindungen zwischen Ereignissen und Funktionen.
Kontrollfluss		Beschreibt die zeitlich-sachlogischen Abhängigkeiten von Ereignissen und Funktionen.
Prozesswegweiser		Zeigt als Navigationshilfe die Verbindung zu einem bzw. von einem anderen Prozess.
Organisatorische Einheit		Beschreibt ein Element der Gliederungsstruktur eines Unternehmens.
Informationsobjekt		Ein Informationsobjekt ist eine Abbildung eines Gegenstandes der realen Welt (z.B. Geschäftsobjekt, Entität)
Informationsfluss		Beschreibt, ob von einer Funktion gelesen oder geschrieben wird.
Zuordnung von Systemorganisationseinheiten		Die Zuordnung beschreibt, welche Einheit (Mitarbeiter) die Funktion bearbeitet.

Tab. 5.1: Elemente einer Ereignisgesteuerten Prozesskette ([Kel99] S.161)

Bei den Verknüpfungsoperatoren wird zwischen den Operatoren für „und“, „oder“ und „exklusives oder“ unterschieden. Wenn mehrere Pfeile eingehen, darf nur ein Pfeil ausgehen (Verknüpfer), wenn mehrere Pfeile ausgehen, darf nur ein Pfeil eingehen (Verteiler). Über gemeinsame Ereignisse kann man zwischen verschiedenen Prozessen navigieren. Mit dem Konstrukt des Prozesswegweisers sollen solche Verbindungen schnell

ersichtlich sein. Die folgende Abbildung zeigt ein Beispiel einer EPK.

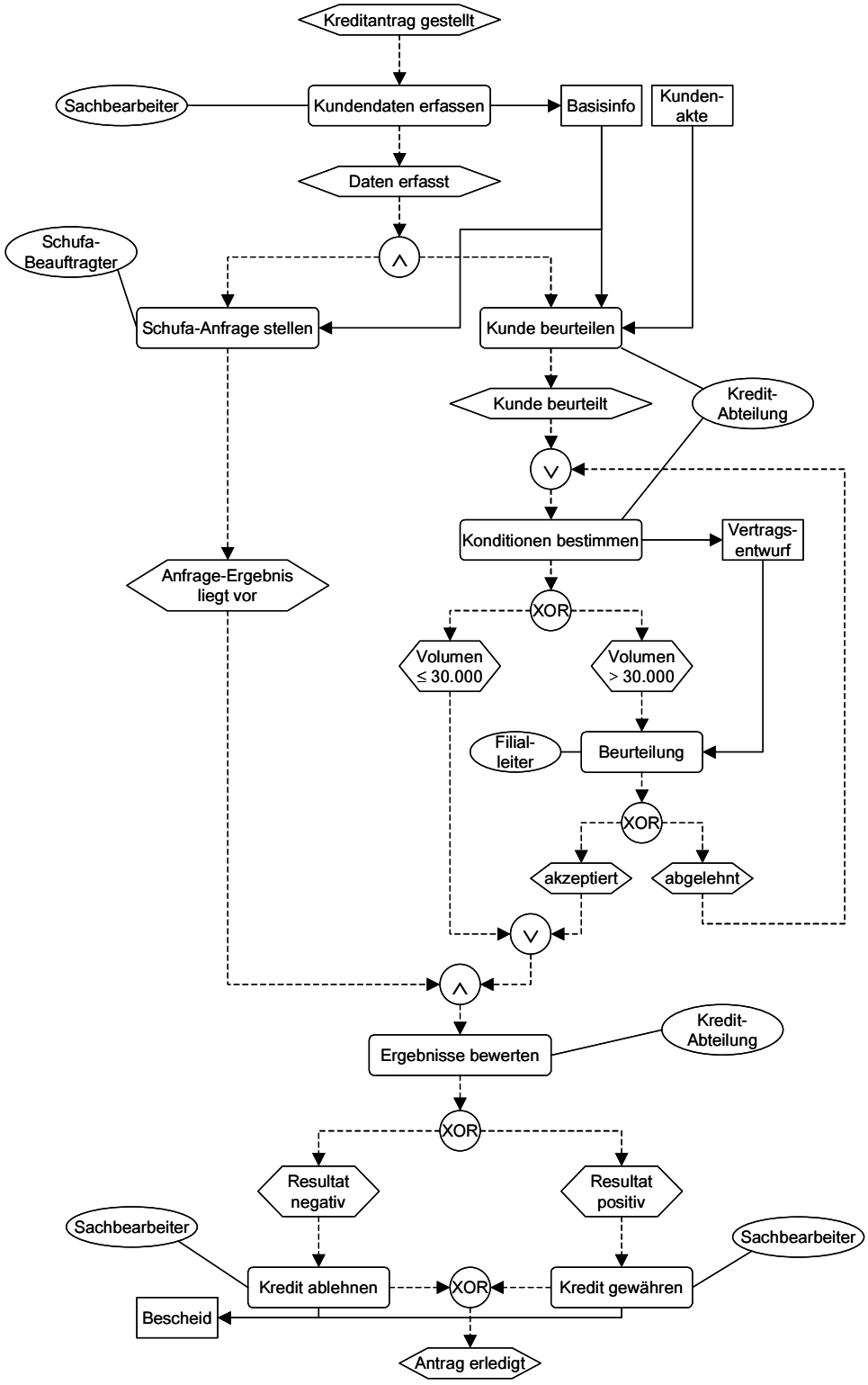


Abb. 5.6: Beispiel für eine Ereignisgesteuerte Prozesskette

5.2.2 Formale Beschreibung der Syntax

Als Schwachpunkt des EPK-Modells in der originalen Fassung (siehe [Kel92]) wird häufig die fehlende oder unvollständige formale Definition von Syntax und Semantik

angegeben. In [Rit00] zum Beispiel heißt es dazu: „Den Vorteilen [..] stehen auch signifikante Nachteile gegenüber, vor allem die Mehrdeutigkeit und Ungenauigkeit der Methode.“ (S.2). Da sich die EPK im Paket mit der betriebswirtschaftlichen Standardsoftware SAP R/3 aber sehr stark verbreitet hat und quasi zu einem Standard in der industriellen Praxis geworden ist, wurde in zahlreichen Arbeiten versucht, das Modell nachträglich durch formale Definitionen zu beschreiben. Solche Ansätze finden sich zum Beispiel in den Veröffentlichungen von P. Rittgen ([Rit99], [Rit00]), der sich dort mit der Problematik der Semantik von Prozessketten auseinandersetzt. In [Kel99] (S.166ff) wird zumindest für die Syntax der EPK eine formale Beschreibung gegeben. Obwohl das EPK-Modell graphbasiert ist, haben sich die Autoren entschieden, die Syntax nicht operativ durch Graphgrammatiken sondern deklarativ durch Spezifikationen anzugeben.

Danach ist ein EPK-Modell (EPKM) ein „typisierter, attributierter, gerichteter, zusammenhängender Graph“, der durch folgendes 7-Tupel gegeben ist:

$EPKM = (Id, \nu, \kappa, \tau, \tau_\kappa, \alpha, \alpha_\kappa)$, dabei ist

Id	die eindeutige Identifizierung des EPKM,
ν	die endliche Menge der Knoten einer EPK, wobei $\lvert\nu\rvert \geq 3$, da jede EPK mindestens ein Startereignis, eine Funktion und ein Endereignis enthält,
κ	die Kantenrelation, die die Verbindungen zwischen den Knoten beschreibt ($\kappa \subseteq \nu \times \nu$),
τ	die Abbildung, die jedem Knoten einen Typ zuweist, $\tau: \nu \rightarrow$ {Funktion, Ereignis, Prozesswegweiser, Oder-Konnektor, ...}
τ_κ	die Abbildung, die jeder Kante einen Typ zuweist, $\tau_\kappa: \kappa \rightarrow$ {Kontrollflusskante, Informationsflusskante, ...}
α	die Abbildung, die jedem Knotentyp seine Attribute zuweist,
α_κ	die Abbildung, die jedem Kantentyp seine Attribute zuweist.

Mit diesem Formalismus lassen sich weitere Begriffe wie Nachbarschaftslisten, Knotengrade und genauere Partitionierungen der Knoten- und Kantenmengen bestimmen. In [Kel99] sind aufbauend auf diesen und weiteren Definitionen Konsistenzbedingungen für ein EPK-Modell formuliert worden, die eine EPK einhalten muss, um bezüglich des Kontrollflusses korrekt zu sein (vgl. [Kel99] S.172f).

5.2.3 Geschäftsprozessmodellierung mit EPK

Mit einer EPK und ihren Konstrukten für Informationsobjekte und organisatorische Einheiten lassen sich Daten, Aufgaben und Organisationen verbinden. Diese Methode bildet „somit das zentrale Element innerhalb des Business Process Engineerings“ ([Kel99] S.160). Die EPK muss dabei aber immer im Zusammenhang mit anderen Teilmodellen des Unternehmens gesehen werden, mit denen etwa die Struktur von Organisation und Daten modelliert werden. So ergibt sich ein umfangreiches Gesamtmodell des Unternehmens. Zur Definition der Zusammenhänge mit anderen Modellen findet sich in [Kel99] (S.164f) der Verweis auf ein Metamodell zur EPK-Struktur, das „eine Übersicht über und ein grundlegendes Verständnis für den Modellaufbau [...] ermöglicht“.

Für eine EPK gibt es verschiedene Darstellungsstrategien auf verschiedenen Abstraktionsstufen. Häufig sieht man zum Beispiel die so genannte „schlanke EPK“, eine Variante, bei der zum besseren Überblick zunächst nur der Kontrollfluss zwischen den Ereignissen und Aufgaben zusammen mit den Verknüpfungsoperatoren beschrieben wird. Die sonst üblichen organisatorischen Einheiten und Informationsobjekte werden dabei vorerst ausgelassen (siehe [Kel99] S.160f).

Zur Analyse und Modellierung eines Unternehmens und der dort auszuführenden Abläufe werden sogenannte Referenzmodelle angeboten. Dabei handelt es sich um Bibliotheken von Prozessmodellen, die von Beratungsunternehmen oder Softwareherstellern wie SAP entwickelt und geliefert werden. SAP stellt zum Beispiel über 800 Prozessbausteine zur Verfügung, mit denen ein Unternehmen, an seine Bedürfnisse angepasst, seine Wertschöpfungsketten modellieren kann. Oft gibt es für bestimmte Branchen bereits spezielle Varianten für nur dort übliche Prozesse. Nach der Auswahl von Prozessbausteinen müssen diese allerdings meistens noch individualisiert und dem jeweiligen Unternehmen angepasst werden. So wird aus dem Referenzmodell ein unternehmensbezogenes Modell (vgl. [Sch98] S.61ff).

Bei der Aufgabenbeschreibung mittels der EPK-Methode ist ein nicht unwichtiges Teilelement die „organisatorische Zuordnung von Aufgaben zu betrieblichen Aufgabenträgern bzw. Unternehmensorganisationseinheiten“ ([Kel99] S.160). Dieser Aspekt wird bei den Modellierungstechniken, die in der vorliegenden Arbeit angewandt werden sollen, allerdings keine wichtige Rolle spielen, weil dort systemorientierte Prozesse betrachtet werden, bei denen Softwarekomponenten und nicht betriebliche Aufgabenträger die Aktionen ausführen.

5.2.4 Eignung für Prozessdiagramme

Die Ereignisgesteuerte Prozesskette ist eine oft benutzte Methode zur Modellierung von Geschäftsprozessen. Inwiefern sie auch den Anforderungen an eine Modellierungssprache zur Lösung der in dieser Arbeit behandelten Thematik genügt, soll im Folgenden durch einen Abgleich mit den Anforderungen aus Abschnitt 3.3 dargelegt werden.

Die Methode der EPK erfüllt die Anforderungen visuell, weitläufig bekannt, ablauforientiert und graphbasiert: Als grafische Modellierungssprache mit zahlreichen symbolischen Repräsentanten für Objekte, Funktionen und Ereignisse erzeugt eine EPK ein sehr anschauliches und eingängiges Bild von dem gewünschten Prozessablauf. Da die Methode im Rahmen der stark verbreiteten betriebswirtschaftlichen Standardsoftware SAP R/3 zur Modellierung von Unternehmensabläufen benutzt wird, ist sie vor allem im Bereich der betriebswirtschaftlichen Anwendung außerordentlich gut bekannt und quasi zu einem Industriestandard geworden (vgl. [Wen00] S.36). Basierend auf dem Prinzip der Vorgangskettendiagramme ist die EPK graphbasiert und ablauforientiert. Ein besonderer Akzent liegt jedoch auf der Steuerung des Ablaufs durch Ereignisse, die jeweils explizit in die Modellierung der Prozesse einfließen müssen. Das Eintreten eines Ereignisses schafft die Voraussetzung für das Anstoßen der nächsten Aufgabe. Bei den in dieser Arbeit anzufertigenden Workflow-Modellen wird das Auftreten von Ereignissen keine so zentrale Rolle spielen. Vielmehr gilt als Auslöse-Ereignis für eine Aktivität in der Regel implizit die Beendigung der Vorgänger-Aktivität(en) (vgl. Abschnitt 10.1).

Zur Modellierung der vier geforderten Kontrollflusskonstrukte Sequenz, Ver-

zweigung, Schleife und Parallelität bietet die EPK den gestrichelten Pfeil und die vorgestellten Verknüpfungsoperatoren AND, OR und XOR an. Die Operatoren OR und XOR dienen in der Regel zur Modellierung von (bedingten) Verzweigungen, der AND Operator zur Modellierung von parallel eintretenden Ereignissen, die verschiedene parallele Teilprozesse auslösen. Die folgende Abbildung zeigt, wie man zwei parallele Teilprozesse durch die AND-Verknüpfung synchronisieren kann.

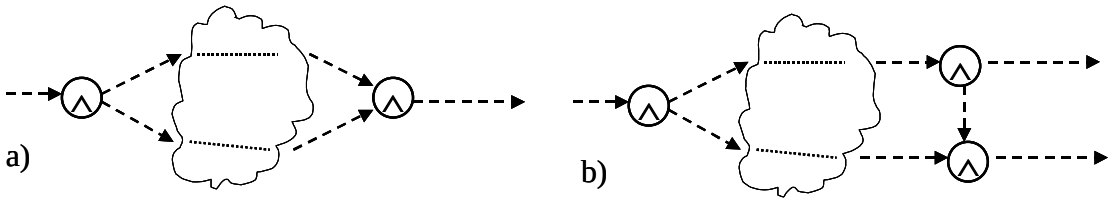


Abb.5.7: Synchronisation von parallelen EPK-Strängen

Im Fall (a) synchronisieren sich beide Parallelprozesse und werden wieder zu einem Strang verschmolzen. Fall (b) zeigt einen Modellausschnitt, bei dem der untere Strang mit der Ausführung an dem AND-Operator wartet, bis der obere Teilprozess den Kontrollpunkt mit dem AND-Operator erreicht hat. Auf diese oder ähnliche Weise lassen sich Prozesssynchronisationen mit der EPK-Methode modellieren.

Die in der Anforderungsanalyse geforderte zweite Eigenschaft der eindeutig definierten Start- und Endpunkte kann mit einer EPK erfüllt werden, da besondere Ereignisse als Start- oder Endereignisse angesehen werden können. Allerdings ist es nicht obligatorisch, sich auf ein eindeutiges Start- oder Endereignis festzulegen. So kann eine EPK auch durch mehrere solcher Ereignisse angestoßen bzw. abgeschlossen werden. Zur Modellierung der XML-Prozesse müsste dieser Freiheitsgrad eliminiert werden. Das ist im Allgemeinen kein Problem, denn man könnte bei mehr als einem Startereignis ein übergeordnetes abstraktes Startereignis einführen, das auf die ursprünglichen Startereignisse verzweigt und bei den Endereignissen analog verfahren.

Bei der EPK ist keine Ein- und Ausgabe von XML-Daten vorgesehen, da die Sprache für beliebige Geschäftsprozesse konzipiert wurde und nicht die Besonderheiten einer XML-Unterstützung berücksichtigt. Somit fehlen zunächst Konstrukte für den Zugriff auf XML-Inhalte sowie die Beschreibung ihrer Struktur. Insbesondere ist es nicht so einfach möglich, den Kontrollfluss in Abhängigkeit der Daten zu lenken. Die Steuerung des Kontrollflusses durch Bedingungen ist nur durch Definition und Einfügen entsprechender Ereignisse möglich; die Verknüpfung von Kontrollflusskanten mit logischen Ausdrücken wie den aus UML-Aktivitätendiagrammen bekannten *Guards* ist nicht vorgesehen (siehe Abschnitt 5.3.1).

Durch die Kernelemente einer EPK werden Ereignisse und Aufgaben zu einem Vorgang miteinander verkettet. Die Forderung nach einer aktivitätenbasierten Modellierung kann also erfüllt werden. Für die Aufgaben in einer EPK können durchaus bestimmte Systemkomponenten vorgesehen werden, so dass ein prozessorientierter Integrationsansatz realisierbar scheint. Üblicherweise werden bei der Modellierung mit einer EPK keine Attribute für die Aufgaben angegeben, obwohl das formale Modell solche Attribute ausdrücklich für die Knoten und Kanten der EPK zulässt. Aufsetzend auf diesen Attributen könnte man die geforderten Parametereinstellungen für die Aktivitäten der EPK realisieren.

Zur Beherrschung der Komplexität von Modellen bietet die EPK drei Alternativen an: die hierarchisierten Funktionen, Prozessmakros und die Prozesswegweiser. Während die Prozesswegweiser in erster Linie zur Verkettung einzelner EPKs genutzt werden, eignet sich die Möglichkeit einer hierarchisierten Funktion zur Realisierung der in der Anforderungsanalyse gewünschten hierarchischen Schachtelung. Bei der Hierarchisierung einer EPK wird in einer Funktion auf einen bereits definierten anderen Prozess verwiesen, der beim Erreichen dieser Aufgabe ausgeführt werden soll (vgl. [Kel99] S.166, 168f).

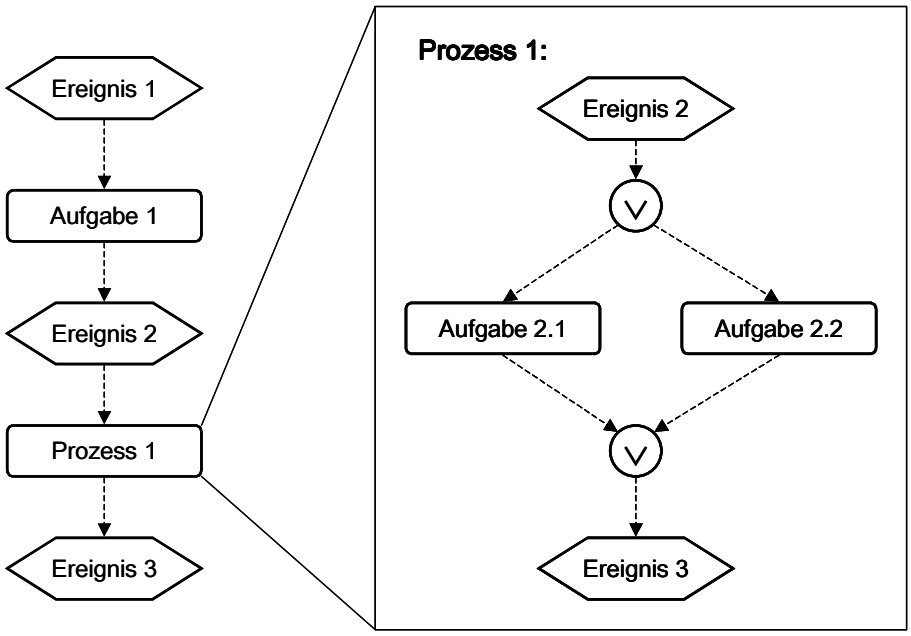


Abb. 5.8: Hierarchisierung einer EPK (nach [Kel99] S.169)

Das Konzept der Hierarchisierung wird auch in den Referenzmodellen von SAP eingesetzt. Dabei werden Bibliotheken mit sogenannten *Prozessbausteinen* gebildet. Jeder Prozessbaustein besteht aus einer elementaren *Prozess-EPK*. Diese Grundbausteine können auf einer zweiten höheren Stufe zu *Szenarien* zusammengefasst werden. Die entsprechende EPK eines Szenarios wird *Szenario-EPK* genannt.

In ARIS gibt es neben der EPK weitere Modelle, mit denen die Struktur des Systems dargestellt werden kann. Diese Sichten könnte man verwenden, um ein statisches Modell zu einem Workflow aufzubauen, in dem die verfügbaren Systemkomponenten für die Funktionen repräsentiert werden. Dabei ist allerdings zu berücksichtigen, dass die statischen Modelle im Umfeld der EPK eigentlich zur Abbildung organisatorischer Einheiten eines Unternehmens eingeführt worden sind und nicht zur Darstellung einer Sammlung von Softwarekomponenten, die innerhalb eines integrativen Workflows angesprochen werden sollen. Dies macht eventuell einige Änderungen an den Modellen nötig.

Die letzte Forderung nach einem Konstrukt zur Modellierung von Transaktionen ist bislang von der EPK-Theorie nicht erfüllt. Dies könnte daran liegen, dass Transaktionen vor allem im Bereich der Ausführung von Softwareprozessen eine Rolle spielen und weniger bei betriebswirtschaftlichen Prozessen. Um diese Anforderungen dennoch zu erfüllen, müsste ein geeignetes Element erst neu definiert und eingeführt werden.

5.3 UML-Aktivitätendiagramme

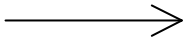
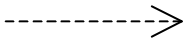
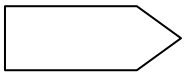
Die Unified Modeling Language (UML) ist eine von der Object Management Group (OMG) standardisierte, grafische Sprache zur Visualisierung und Modellierung der Eigenschaften objektorientierter Softwaresysteme. Die UML-Spezifikation beschreibt verschiedene Diagrammarten, die sich für unterschiedliche Anwendungsgebiete eignen. Aktivitätendiagramme sind eine von mehreren Möglichkeiten, dynamische Abläufe darzustellen. Laut [Oes97] sind in ihnen Ansätze einiger anderer Sprachen vereinigt, darunter Ereignisdiagramme, Zustandsdiagramme und Petri-Netze. In den folgenden Abschnitten sollen Aktivitätendiagramme vorgestellt und ihre Eignung für die Modellierung von Geschäftsprozessen untersucht werden.

5.3.1 Basiskonstrukte von Aktivitätendiagrammen

Ein Aktivitätendiagramm ist laut UML-Spezifikation (siehe [Omg01] 3-161) eine spezielle Zustandsmaschine, in dem Zustände als *Aktivitäten* interpretiert werden können. Eine Aktivität wird grafisch durch einen Rahmen dargestellt, bei dem obere und untere Kante gerade Linien und linke und rechte Kante konvexe Bögen sind (vgl. Tabelle 5.2). Die Aktivitäten sind verbunden durch *Transitionen,* welche als Pfeile dargestellt werden. Dabei hat jede Aktivität mindestens je eine Eingangs- und Ausgangsaktivität. Die Ausführung einer Aktivität wird ausgelöst, sobald alle ihre Eingangsaktivitäten beendet sind. Zur besseren Übersichtlichkeit der Diagramme ist es möglich, komplette Teildiagramme in eine Aktivität des übergeordneten Diagramms einzubetten bzw. darauf zu verweisen. Das Teildiagramm wird dann ausgeführt, wenn die Ausführung des übergeordneten Diagramms bei der entsprechenden Aktivität ankommt. Man spricht dabei auch von *Subaktivitätszuständen*.

Zusätzlich zu den beiden grundlegenden Elementen Aktivität und Transition besitzen Aktivitätendiagramme eine Reihe weiterer Sprachelemente, um spezielle Situationen im Kontrollfluss zu beschreiben. Mit Hilfe von Verzweigung und Vereinigung ist es möglich, abhängig von Bedingungen unterschiedliche Ausführungswege zu wählen, die sich später wieder zu einem einzigen Strang vereinigen. Die Bedingungen für die einzelnen Wege werden als sogenannte *Guards* in eckigen Klammern an den Ausgangstransitionen der Verzweigung notiert.

Um Nebenläufigkeit in einem Aktivitätendiagramm auszudrücken, steht der schwarze Balken zur Verfügung. Wenn er eine Eingangs- und mehrere Ausgangstransitionen besitzt (Fork-Zustand), werden mehrere Ausführungsstränge gestartet, die parallel abgearbeitet werden. Sie können durch einen schwarzen Balken wiedervereinigt werden, in den alle Stränge hineinlaufen, aber nur eine Ausgangstransition existiert (Join-Zustand). Eine solche Stelle dient zugleich als Synchronisationspunkt, da die Ausführung erst dann fortfährt, wenn alle parallelen Stränge am schwarzen Balken angekommen sind.

Bezeichnung	Symbol	Definition
Aktivität		Zustand, in dem eine bestimmte Aktivität ausgeführt wird.
Transition		Beschreibt den Kontrollfluss zwischen den einzelnen Aktivitäten des Diagramms.
Verzweigung/ Vereinigung		Beschreibung einer Verzweigung des Kontrollflusses in Abhängigkeit von bestimmten Bedingungen (Guards).
Nebenläufigkeit		Erlaubt die Erzeugung und Synchronisation von parallelen Ausführungssträngen.
Synchronisations-zustand		Ermöglicht die Synchronisation zwischen verschiedenen parallelen Ausführungssträngen.
Bahnen		Dient der Beschreibung, welche Systemkomponente welche Aktivitäten ausführt.
Objekt		Darstellung eines Objekts, das von einer Aktivität gelesen oder geschrieben wird.
Objektfluss		Beschreibung des Datenflusses eines Objekts zwischen verschiedenen Aktivitäten.
Signal senden		Spezielle Aktivität, die zum Senden eines Signals an eine andere Aktivität dient.
Signal empfangen		Spezielle Aktivität, die ein Signal von einer anderen Aktivität empfängt.

Tab. 5.2: Elemente eines UML-Aktivitätendiagramms

Um explizit mehrere Ausführungsstränge zu synchronisieren, dienen Synchronisationszustände. Dabei kann eine ganze Zahl als Schranke für die Differenz zwischen der Anzahl ausgelöster eingehender und ausgehender Transitionen angegeben werden oder ein Stern für eine unbeschränkte Differenz (für Details siehe [Omg01] 2-160).

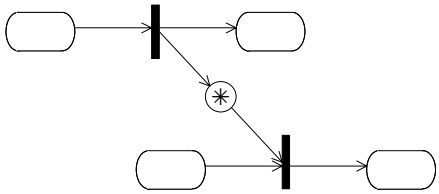


Abb. 5.9: Synchronisation von parallelen Teilprozessen

Häufig möchte man zusätzlich zum Kontrollfluss auch den Datenfluss in einem Aktivitätendiagramm modellieren. Dazu können Objekte in das Diagramm eingefügt werden. Von derjenigen Aktivität, die das Objekt erzeugt oder ändert, wird ein ausgehender Objektfluss-Pfeil zum Objekt gezogen. Alle Aktivitäten, die das Objekt lesen, erhalten einen eingehenden Pfeil.

Es gibt darüber hinaus einige Sprachelemente, die seltener zum Einsatz kommen, sie seien hier jedoch kurz erwähnt. Zum einen könnte sich die Frage stellen, welche Systemkomponente für die Ausführung einer bestimmten Aktivität zuständig ist. Um dies im Diagramm darzustellen, verwendet man vertikale *Bahnen* (engl.: swimlanes), von denen jede genau einer ausführenden Komponente zugeordnet ist. Alle Aktivitäten, die von dieser ausgeführt werden, erscheinen dann innerhalb der entsprechenden Bahn. Des Weiteren gibt es in Aktivitätendiagrammen Elemente, um explizit das Senden und Empfangen von Signalen auszudrücken. Laut [Omg01] (3.91 und 3.91.2) lässt sich dies jedoch auch durch gewöhnliche Aktivitäten realisieren.

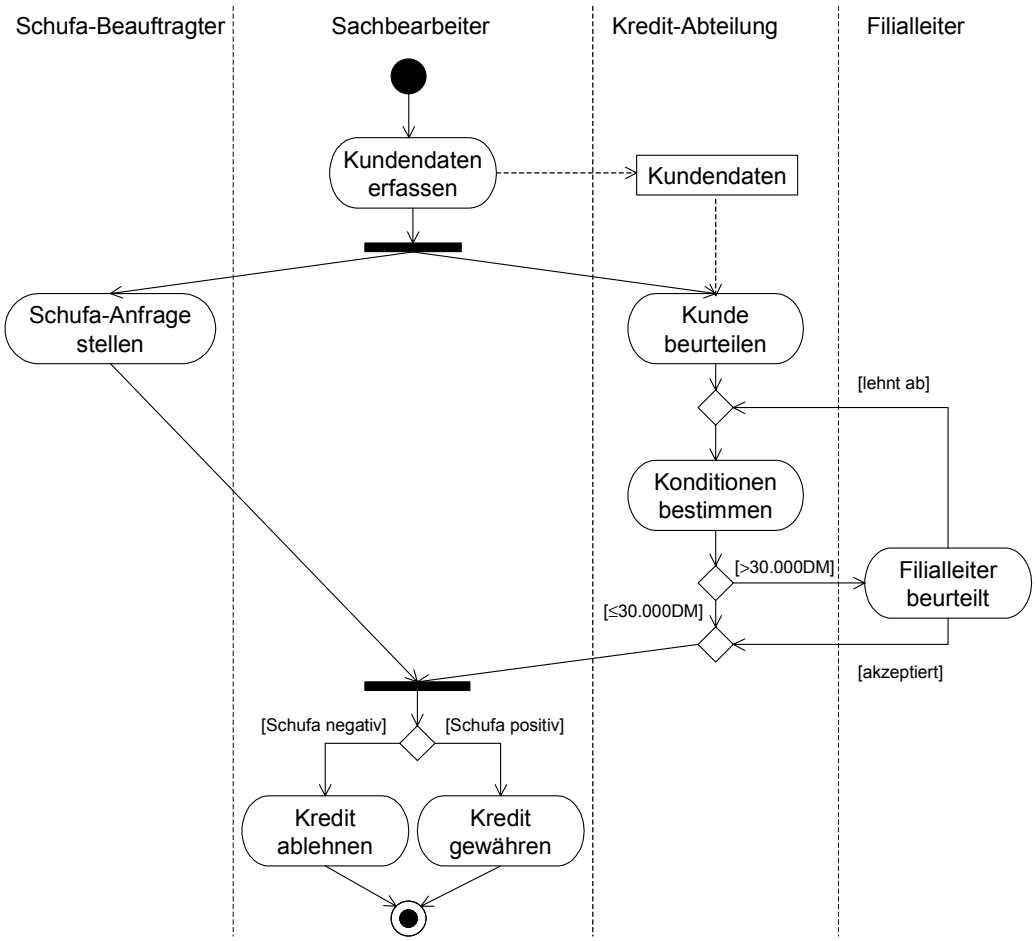


Abb. 5.10: Beispiel eines UML-Aktivitätendiagramms

5.3.2 Geschäftsprozessmodellierung mit UML

Die UML enthält neben Aktivitätendiagrammen eine Reihe weiterer grafischer Sprachen, die sich für die Modellierung von Geschäftsprozessen unterschiedlich gut eignen (vgl. [Oes97]). *Use-Case-Diagramme* dienen im Allgemeinen dazu, die Sicht eines externen Benutzers auf ein System zu modellieren. Dies geschieht in Form von Anwendungsfällen, die beschreiben, welche Aktionen von außen am System ausgelöst werden können. Im Bereich von Geschäftsprozessen eignen sich Use-Case-Diagramme, um in einem ersten Schritt alle Geschäftsprozesse zu erfassen, die in einem System vorkommen. Dabei lassen sich auch die Beziehungen der Prozesse zu den beteiligten Akteuren sowie zu anderen Prozessen beschreiben. Auf diese Weise kann man den gesamten Kontext eines Geschäftsprozesses modellieren.

Als nächster Schritt nach der Erfassung aller Prozesse durch Use-Case-Diagramme muss ein statisches Modell der beteiligten Objekte erstellt werden. Dies können aktive Personen oder Systemkomponenten ebenso sein wie Daten, die innerhalb des Prozesses erzeugt oder verarbeitet werden. In UML bieten sich dafür *Klassendiagramme* an, in denen sich nicht nur Eigenschaften der einzelnen Objekte, sondern auch ihre Beziehungen untereinander beschreiben lassen. Klassendiagramme sind somit ein geeignetes Mittel für die Daten- und Organisationssicht (siehe [Sch98] S.36) auf einen Geschäftsprozess.

Für die Modellierung der dynamischen Aspekte eines Geschäftsprozesses kennt die UML mehrere Diagrammarten. Sowohl *Sequenz-* als auch *Kollaborationsdiagramme* veranschaulichen den Nachrichtenaustausch zwischen den beteiligten Objekten. Während jedoch in Sequenzdiagrammen die zeitliche Reihenfolge im Vordergrund steht, lassen sich durch Kollaborationsdiagramme besonders gut die Beziehungen der Objekte zueinander ausdrücken. Beide Diagrammarten haben den Nachteil, dass sie bei komplexen Abläufen groß und unübersichtlich werden. Im Rahmen der Geschäftsprozessmodellierung ist ihre Aufgabe daher eher darin zu sehen, einzelne Ausschnitte eines Prozesses zu verdeutlichen, die besonders kompliziert oder schwer verständlich sind.

Aufgrund ihrer netzartigen Struktur und der Möglichkeit, Aktivitäten durch Transitionen flexibel miteinander zu verketten, sind Aktivitätsdiagramme in besonderer Weise dafür geeignet, verschiedenste Abläufe zu modellieren. Sie werden in der Regel das bevorzugte Mittel sein, um die dynamische Sicht auf einen Geschäftsprozess zu beschreiben. Ihre vielfältigen Sprachkonstrukte reichen für die meisten denkbaren Situationen aus. Eine besondere Stärke von Aktivitätendiagrammen liegt im Umgang mit parallelen Abläufen, die häufig in Geschäftsprozessen vorkommen.

Die UML-Spezifikation [Omg01] erlaubt es, in Aktivitätszuständen oder Guard-Bedingungen informelle Angaben über die auszuführende Aktion zu machen. Aus diesem Grund lassen sich Aktivitätendiagramme sowohl auf einer hohen, konzeptionellen Ebene einsetzen, als auch sehr implementierungsnah als Darstellung konkreter Programmabläufe. Es ist jedoch zunächst nicht möglich, die Einhaltung fester Rahmenbedingungen für den Inhalt einer Aktivität zu fordern. Diese Ungenauigkeit führt zu einer nicht exakt festgelegten Semantik der Diagramme, was ein häufig genannter Nachteil von Aktivitätendiagrammen ist (siehe z.B. [Esh01] S.33 und [Dum01] S.1). Da eine eindeutig definierte Semantik aber die Grundvoraussetzung für die maschinelle Kontrolle und Ausführung von Geschäftsprozessen durch Workflow-Management-

Systeme ist, wurden unterschiedliche Versuche vorgenommen, um Aktivitätendiagramme formal zu definieren. In [Esh01] beispielsweise werden zunächst Bedingungen erarbeitet, die von Aktivitätendiagrammen zu erfüllen sind, wenn es sich um die Beschreibung eines Workflows handeln soll. Es wird dann eine formale Definition der Syntax entwickelt, die ein Aktivitätendiagramm als Hypergraph auffasst. Auf der Basis dieser Definition erfolgt eine Formulierung der Bedingungen an ein gültiges Workflow-Diagramm. Es wird eine formale Beschreibung von Datenmanipulationen in den Zuständen entwickelt, die schließlich dazu benutzt wird, die komplette Ausführungssemantik von Diagrammen exakt festzulegen.

Ein weiterer Nachteil von Aktivitätendiagrammen ist es, dass sich im Zusammenhang mit Workflow-Modellen die Verknüpfung mit der Organisationssicht schlecht ausdrücken lässt. Man kann zwar mit Hilfe von Bahnen grafisch darstellen, wer oder was bestimmte Aktivitäten ausführt, daraus leitet sich aber keine feste semantische Bedeutung ab (siehe [Esh01] S.12). Insbesondere lässt sich auf diese Weise kein Rollen- bzw. Rechtekonzept realisieren (siehe [Oes97]). Es können zwar die Aktivitäten mit Hilfe von Bahnen einzelnen Rollen zugeordnet werden, sowohl eine Überlappung von Rollen als auch die Zuordnung konkreter Akteure zu Rollen lässt sich aber nicht ohne Weiteres modellieren. In komplexen Workflow-Modellen können Bahnen die gewünschte Zuordnung von Aktionen zu Organisationseinheiten daher nicht leisten.

5.3.3 Eignung für Prozessdiagramme

Im Abschnitt 3.3 wurden Anforderungen aufgestellt, die eine Sprache erfüllen muss, um damit XML-Prozesse modellieren zu können. Bei Aktivitätendiagrammen handelt es sich um eine visuelle Sprache zur Beschreibung und Darstellung von dynamischen Abläufen. Seit die UML durch eine offizielle Spezifikation standardisiert wurde (siehe [Omg01]), ist aus Aktivitätendiagrammen ein Quasi-Industriestandard im Bereich der Softwareentwicklung geworden. Damit sind die Grundvoraussetzungen für einen Einsatz bei XML-basierten Workflow-Modellen gegeben.

Des Weiteren sind alle vier grundlegenden Kontrollflusskonstrukte vorhanden. Es können problemlos sequentielle und verzweigte Abläufe, sowie Schleifen beschrieben werden, wobei Schleifen auf Verzweigungen abgebildet werden. Für die Behandlung von Parallelität verfügen Aktivitätendiagramme über flexible Möglichkeiten, die sowohl das Erzeugen und Wiedervereinigen von parallelen Teilsträngen erlauben, als auch die zwischenzeitliche Synchronisation an bestimmten Ausführungspunkten.

Prinzipiell ist es in Aktivitätendiagrammen erlaubt, die Ausführung an mehreren Stellen in verschiedenen Systemzuständen zu beenden. Mit Hilfe eines grafischen Werkzeugs für das Erstellen von Diagrammen ist es aber leicht sicherzustellen, dass nur je eine Start- bzw. Endaktivität existiert. Es müssen lediglich alle Systemzustände, in denen die Abarbeitung enden soll, durch Transitionen mit einem neuen, eindeutigen Endzustand verbunden werden.

Aktivitätendiagramme bieten die Möglichkeit, Objektflüsse zwischen einzelnen Aktivitäten zu spezifizieren. Dies ist ein Ansatzpunkt für die Verarbeitung von XML-Dokumenten durch die Aktionen eines Diagramms. Zusätzlich wurde gefordert, dass für jede Stelle innerhalb eines Prozessmodells Angaben über die Struktur des XML-Dokuments gemacht werden können (Abschnitt 3.3 Punkt 3). Solche Gültigkeitsaussagen sind bislang nicht vorgesehen.

Die UML-Spezifikation erlaubt verschiedene Möglichkeiten, um den Inhalt von Aktivitäten zu spezifizieren. Da in XML-Prozessen jede Aktivität eine Systemkomponente repräsentiert, die XML-Daten verarbeiten und ggf. manipulieren kann, muss hier eine Einschränkung definiert werden, damit nur zugelassene Komponenten verwendet werden können. In diesem Zusammenhang muss eine Möglichkeit geschaffen werden, Parametereinstellungen an diesen Komponenten vorzunehmen, die bei der Ausführung der zugehörigen Aktivität berücksichtigt und an die Komponente übergeben werden.

Die Möglichkeit, Prozesse durch Bausteine aggregieren bzw. in Teilprozesse aufspalten zu können, ist durch das Konstrukt der Subaktivitätszustände gegeben, in denen ein komplettes Teildiagramm in ein übergeordnetes integriert werden kann. Die letzte Forderung nach einem Konstrukt zur Spezifizierung von Aktionsfolgen als Transaktion ist in Aktivitätendiagrammen bisher nicht umgesetzt worden.

5.4 Bewertung

Nachdem in den vorangegangenen Abschnitten die ausgewählten Modellierungssprachen ausführlich gegenüber den einzelnen Anforderungen evaluiert wurden, soll nun ein zusammenfassendes Fazit gezogen werden. Dazu sind in Tabelle 5.3 alle Anforderungen aus Abschnitt 3.3 bzw. 3.4 und die Bewertung der einzelnen Sprachen in Kurzform aufgelistet.

Anforderung	**Petri-Netze**	**EPK**	**UML-Aktiv.**
Visuelle Sprache	+	+	+
Allgemein akzeptiert	o	+	+
Leicht/Intuitiv verständlich	-	o	+
Ablauforientiert	+	+	+
Alle Basiskontrollfluss-Konstrukte	+	+	+
Ein Start- und ein Endpunkt	1)	1)	1)
XML-Dokumentfluss	2)	2)	2)
Zugriff auf XML-Dokumente	2)	-	2)
Integration von Softwarekomponenten	-	+	+
Parametrisierung der Komponenten	-	-	+
Geschachtelte Prozesse (Hierarchie)	+	+	+
Statisches Modell der Komponenten	-	+	+3)
Transaktionale Prozesse	-	-	-
Präzise/Eindeutige Semantik	+	-	o

1) Bisher nicht gefordert, aber im Rahmen von Werkzeugunterstützung leicht sicherzustellen.
2) Nicht implizit vorgesehen, aber mit vorhandenen Mitteln realisierbar.
3) Mit Hilfe von UML-Klassendiagrammen.

+ = erfüllt - = nicht erfüllt o = teilweise erfüllt

Tab. 5.3: Bewertung vorhandener Workflow-Modellierungssprachen

Zunächst scheint es, dass sich Petri-Netze aufgrund ihrer exakt definierten Semantik zur Modellierung und Ausführung von XML-Prozessen eignen. Es ergeben sich jedoch einige Nachteile, die gegen ihre Verwendung sprechen: Petri-Netze neigen dazu, schnell komplex und unübersichtlich zu werden (siehe [Böh95] S.12). Das liegt zum einen daran, dass jeder Zustandsübergang durch einen eigenen Knoten (Transition) und mindestens zwei Kanten dargestellt wird. Damit besitzen die Netze etwa doppelt so viele Knoten wie Diagramme vergleichbarer Modellierungssprachen, die lediglich eine gerichtete Kante als Zustandsübergang verwenden. Zum anderen führen Erweiterungen der klassischen Petri-Netze häufig zu textuellen Ergänzungen oder Anmerkungen innerhalb der grafischen Darstellung (siehe [Wen00] S.39). Sowohl die große Anzahl an Knoten als auch textuelle Beschriftungen widersprechen dem Ziel der leichten Verständlichkeit und Lesbarkeit der Modelle. Weiterhin ergeben sich bei XML-Prozessen spezielle Anforderungen, wie der Umgang mit XML-Dokumenten oder die Integration externer Softwarekomponenten. Die Umsetzung solcher Anforderungen durch eine formale Erweiterung von Petri-Netzen wäre nicht ohne Weiteres möglich. Das Konzept der XML-Netze hierzu kann ebenfalls nicht verwendet werden. Das entscheidende Argument gegen den Einsatz von Petri-Netzen ist jedoch, dass sie zwar weitläufig bekannt sind, sich aber in der Praxis nie durchsetzen konnten. Laien empfinden sie häufig als zu technisch und unhandlich.

Die Ereignisgesteuerten Prozessketten sind in der Betriebswirtschaft ein bewährtes Mittel zur Modellierung betrieblicher Abläufe und Geschäftsprozesse. Besonders wegen ihrer recht einfachen und leicht verständlichen Darstellungsweise eignen sie sich für die Anwendung durch technisch nicht geschultes Personal, das damit Prozesse auf hohem Niveau modellieren kann. Dieser Vorteil wird allerdings um den Preis einer fehlenden oder nicht ausreichenden formalen Festlegung von Semantik der Modelle erkauft. Zwar haben einige wissenschaftliche Arbeiten in dieser Richtung Verbesserungen und Ergänzungen gebracht, doch gibt es immer noch eine semantische Lücke zwischen dem Geschäftsprozessmodell und einer Spezifikation, die von einem Workflow-Management-System ausgeführt werden könnte. Ziel dieser Arbeit ist es aber, ein Konzept aufzuzeigen, mit dem XML-Prozesse so modelliert werden können, dass sie anschließend automatisch ausgeführt werden können. Mit den semi-formalen Ereignisgesteuerten Prozessketten ließe sich dieses Ziel ohne Verfeinerung des Modells nicht erreichen.

Außerdem ist abzuwägen, inwiefern die Fokussierung auf die Ereignissteuerung bei einer EPK für XML-Prozesse überflüssig oder vielleicht sogar störend sein könnte. Sind Ereignisse für den Entwurf von Prozessdiagrammen nicht unbedingt nötig und würden sie nur eingefügt, weil das Modell der EPK diese fordert, könnte die Darstellung unnötig kompliziert und aufgebläht werden. Dies hätte negative Einflüsse auf die Überschaubarkeit und Verständlichkeit der Entwürfe. Trotz der Erfolge des EPK-Modells im Bereich der Geschäftsprozessmodellierung ist es daher zweifelhaft, ob sie die beste Grundlage zum Entwurf XML-basierter Workflow-Modelle sind.

UML-Aktivitätendiagramme sind ein geeignetes Mittel zur visuellen Beschreibung von nahezu beliebigen dynamischen Abläufen in komplexen Systemen. Im Rahmen der Verbreitung der UML haben sie sich sowohl im wissenschaftlichen als auch im praktischen Bereich als weithin bekannter Standard etabliert. Im Vergleich mit anderen Modellierungssprachen wie Petri-Netzen und Ereignisgesteuerten Prozessketten scheinen Aktivitätendiagramme die übersichtlichste Variante zu sein. Sie sind daher auch für

technisch weniger kundige Personen verständlich. In Bezug auf die Modellierung von Workflows im Allgemeinen und XML-Prozesse im Speziellen hat sich gezeigt, dass Aktivitätendiagramme fast alle notwendigen Sprachkonstrukte besitzen (vgl. [Dum01] S.1) und ihre Funktionsweise mit der Idee von XML-Prozessen harmoniert. An einigen Stellen werden Erweiterungen erforderlich sein, die sich aber mit Hilfe der UML-Erweiterungsmechanismen (siehe Abschnitt 7.2) auf standardisierte Weise vornehmen lassen. Wir haben uns daher entschieden, als Ausgangspunkt für die Modellierung mit Prozessdiagrammen UML-Aktivitätendiagramme zu wählen. Im folgenden Kapitel wird eine Workflow-Modellierungssprache auf Basis von UML-Aktivitätendiagrammen vorgestellt, die die notwendigen Erweiterungen enthält.

6 XML-Prozesse und Prozessdiagramme

Nach der vorausgegangenen Evaluierung ausgewählter Systeme und Modellierungssprachen möchten wir nun unseren eigenen Ansatz für XML-basierte Workflows vorstellen. Dabei geht es vor allem um das zu Grunde gelegte Konzept für XML-Prozesse und die Vorstellung von Prozessdiagrammen als Sprache, mit der sich solche Workflows modellieren lassen. Die Umsetzung als Workflow-Management-System wird in einem Architekturvorschlag nur grob skizziert und in den Kapiteln 9 und 10 näher beschrieben.

6.1 Konzeption von XML-Prozessen

In diesem Abschnitt erläutern wir grundlegende Konzepte zur Umsetzung der Anforderungen an XML-Prozesse aus Kapitel 3. Im Einzelnen befassen sich die folgenden Unterabschnitte mit den Anforderungen an

- die Behandlung Workflow-relevanter Daten in Form eines Eingabedokumentes,
- die Verarbeitung der Dokumente,
- die flexible Einbindung von bestehenden Softwarekomponenten,
- die dynamische Bindung von Komponenten an Modellelemente und
- die Rolle eines Interpreters zur Kontrolle und Ausführung der Prozessinstanzen.

6.1.1 Dokumentmigration

Wie in der Fallstudie aus Abschnitt 3.1 aufgezeigt, werden elektronische Geschäftsvorgänge in der Regel durch den Eingang eines bestimmten Dokumentes ausgelöst. Dieses Dokument repräsentiert eine Anfrage an die E-Business-Plattform und muss von dieser in geeigneter Weise bearbeitet werden, um die Anfrage zu beantworten und die gewünschten Informationen zu beschaffen oder zu berechnen. Im Referenzmodell der Workflow Management Coalition werden die Daten eines solchen Dokuments, die dem WFMS bei der Initialisierung eines Prozesses übergeben werden müssen, als *Workflow-relevante Daten* bezeichnet (siehe [Wmc95] S.14/26). Die Beantwortung der Anfrage nach der Verarbeitung des Dokumentes erfolgt durch die Rückgabe der Ergebnisdaten, zum Beispiel an einen Web-Browser oder ein anderes, vom Klienten eingesetztes Endgerät. Diese Rückgabedaten können gleichfalls wie das Paket der Eingabedaten als Dokument aufgefasst werden.

Die Auslösung eines Vorgangs durch das Eintreffen eines Dokumentes, das gegebenenfalls den nötigen Vorgangstyp selektiert und weitere Parameterwerte für die Durchführung des Vorgangs enthält, sowie die Ausgabe eines Ergebnisdokumentes nach Beendigung des Vorgangs führt uns zu einem *Dokumentmigrationsmodell* für die Ausführung von Workflow-Modellen, wie es in der folgenden Grafik dargestellt ist.

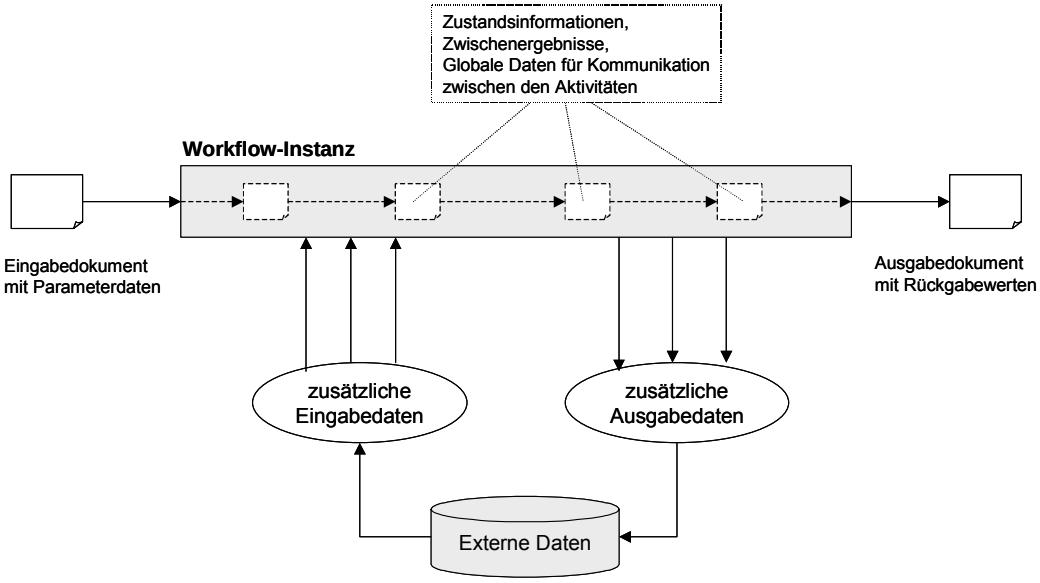


Abb. 6.1: Dokumentmigrationsmodell für die Workflow-Ausführung

Bei dem Dokumentmigrationsmodell wird von der Eingabe eines Dokuments in einem strukturierten Datenformat ausgegangen. Dieses Eingabedokument wird vom Workflow-Management-System entgegengenommen und verarbeitet. Dazu klassifiziert das WFMS den Dokumenttyp und erzeugt für diesen Typ eine Instanz des zu seiner Verarbeitung nötigen Workflow-Modells.

Basierend auf den im Bereich des Workflow-Managements bekannten *Objektmigrationsmodellen* wollen wir bei diesem Dokumentmigrationsmodell davon ausgehen, dass das Eingangsdokument wie eine Umlaufmappe in einer Behörde den gesamten Workflow durchläuft und von Aktivität zu Aktivität weitergereicht wird. Bei den Objektmigrationsmodellen (vgl. [Wen00] S.32) werden die Dokumente bei ihrer Modellierung bereits mit allen steuerungsrelevanten Informationen ausgestattet, so dass sie selbstständig die Weiterleitung zum nächsten Sachbearbeiter veranlassen können. Dadurch geht die Kontrolle und Steuerung durch eine zentrale Managementkomponente verloren. Um das zu verhindern, werden die Dokumente bei unserem Ansatz nicht mit impliziten Informationen über den Ablauf ihrer Verarbeitung ausgestattet. Diese Informationen befinden sich vielmehr in dem jeweiligen Workflow-Modell, das zur Verarbeitung des Dokumentes herangezogen und von einem zentralen Interpreter ausgeführt wird.

Trotzdem lassen sich bei den Dokumentmigrationen Analogien zu dem Prinzip der Umlaufmappen erkennen, das in [Böh95] (S.9) vorgestellt wird. Diese früher in Büros üblichen Mappen enthalten alle Informationen über den aktuellen Stand eines Vorgangs sowie alle für diesen Vorgang relevanten Dokumente. Jeder Sachbearbeiter kann Dokumente bearbeiten oder die Mappe um neue Dokumente ergänzen. Nach Erledigung seines Anteils an der Vorgangsbearbeitung leitet er die Mappe an einen anderen Kollegen oder eine andere Instanz weiter, die dann mit dem nächsten Schritt der Bearbeitung fortfährt, bis schließlich der gesamte Vorgang erledigt ist.

Ein ähnlicher Gedanke liegt dem Dokumentmigrationsmodell zu Grunde: Das Eingangsdokument wird vom WFMS entgegengenommen und in ein internes, strukturiertes Datenformat transformiert. Von diesem Zeitpunkt an repräsentiert das Dokument

die Umlaufmappe, die vom WFMS als zentraler Steuerungskomponente von Aktivität zu Aktivität weitergereicht wird. Jede Aktivität kann zu ihrer Erledigung nötige Informationen aus dem Dokument entnehmen, sowie die Ergebnisse der Aktivität als Zwischenergebnis und als Eingabe für nachfolgende Aktivitäten in dem Dokument ablegen. Die aus dem Dokument entnommenen oder vom WFMS aus dem Workflow-Modell abgeleiteten Informationen und Parameter müssen ausreichen, um die Ausführung der einzelnen Aktivität zu spezifizieren. Neben diesen Startinformationen kann jede Aktivität weitere Eingabedaten aus externen Datenquellen beziehen. Solche Daten, auf die das WFMS selbst keinen Zugriff hat, werden im Workflow-Referenzmodell *Workflow-Applikationsdaten* genannt (siehe [Wmc95] S. 14). Handelt es sich dabei zum Beispiel um das Ergebnis einer Datenbankanfrage, benötigt die Instanz, die die Aktivität ausführen soll, Schlüsselattribute, um die richtigen Datensätze aus den Tabellen der Datenbank zu selektieren. Solche Schlüsselinformationen befinden sich in der Regel in dem Eingabedokument des Workflows. Ein typisches Beispiel ist etwa die Kundennummer eines Klienten, der für seine Konten eine bestimmte Funktion ausführen lassen möchte.

Genauso wie das Einlesen aus externen Datenquellen möglich ist, können in Aktivitäten auch Daten in externen Datenspeichern abgelegt werden. Auf diese Weise können Aktivitäten unabhängig vom umlaufenden Dokument beliebige Ausgabedaten produzieren. Alle für den Fortgang des Workflows und für die nachfolgenden Aktivitäten wichtigen Informationen sollten jedoch in dem Migrationsdokument abgelegt werden. So kann eine Aktivität auch Nachrichten in dem Dokument ablegen, die von einer nachfolgenden Aktivität interpretiert werden können. Das verwendete strukturierte Datenformat sollte es auch erlauben, neue Teildokumente in das Migrationsdokument einzufügen, so, wie ein Sachbearbeiter ein neues Dokument der Umlaufmappe hinzufügt.

Das Migrationsdokument hat somit mehrere Funktionen. Es dient zur:

- Eingabe der für die Ausführung des Workflows nötigen Parameter,
- Rückgabe der Ergebnisse nach der Workflow-Ausführung,
- Kommunikation einer Aktivität mit nachfolgenden Aktivitäten,
- Sammlung von Zwischenergebnissen, die schließlich zum Endergebnis zusammengesetzt werden können sowie zur
- Repräsentation des augenblicklichen Zustandes des Workflows.

Nach der vollständigen Ausführung sollte das Migrationsdokument alle nötigen Informationen und Daten zur Beantwortung der Anfrage, also das Ergebnis der Workflow-Ausführung enthalten. Vor der Rückgabe dieses Ergebnisses ist gegebenenfalls noch eine Rücktransformation aus dem intern verwendeten, strukturierten Datenformat in das gewünschte Zielformat nötig.

Dokumentmigration im Kontrollfluss versus expliziter Datenfluss

Beim Dokumentmigrationsmodell fallen Kontroll- und Datenfluss zusammen, weil das Migrationsdokument von einer Aktivität entsprechend dem Kontrollfluss der Prozessausführung zur nächsten weitergeleitet wird. Da das Migrationsdokument alle Workflow-relevanten Daten enthält, ist ein zusätzlicher Datenfluss zwischen den einzelnen Aktivitäten eines Workflow-Modells nicht vorgesehen.

Alternativ wäre auch der Ansatz eines expliziten Datenflusses denkbar, der nicht

von der festen Vorgabe eines einzigen Dokumentes ausgeht, sondern die Modellierung und Ausführung von beliebigen Datenflüssen zwischen den Aktivitäten ermöglicht, die weitgehend unabhängig vom Kontrollfluss verlaufen könnten. Dadurch wäre es möglich, die Datenflüsse flexibler zu modellieren, ein Dokument zum Beispiel direkt von einer Aktivität zu einer anderen, nicht unmittelbar nachfolgenden Aktivität zu senden, die Daten in mehrere kleinere Teildokumente aufzuteilen oder für eine Aktivität auch mehrere Ein- oder Ausgabedokumente zu modellieren.

Aus unserer Sicht sind beide Ansätze jedoch weitgehend gleich mächtig und lassen sich ineinander überführen: Die Dokumentmigration etwa könnte durch allgemeinen Datenfluss realisiert werden, indem man sich dort auf nur ein Dokument beschränkt und den Datenfluss synchron zum Kontrollfluss modelliert, so dass dieses Dokument entsprechend dem Kontrollfluss von Aktivität zu Aktivität migriert. Auf der anderen Seite kann durch das Migrationsdokument ein expliziter Datenfluss simuliert werden, weil das Dokument gemäß der Analogie zu den Umlaufmappen als Container für die verschiedenen Teildokumente angesehen werden kann, die im Datenfluss vorkommen. Das verwendete Datenformat (z.B. XML) muss diese Schachtelung und Zusammenfassung von Teildokumenten erlauben. Das Migrationsdokument fließt im Laufe des Kontrollflusses zu allen Aktivitäten, so dass diese die für sie relevanten Informationen entnehmen können.

Obwohl die explizite Berücksichtigung von Datenflüssen bei der Modellierung flexibler ist, haben wir uns für das Dokumentmigrationsmodell entschieden, weil entsprechend der Fallstudie aus Abschnitt 3.1 die Verarbeitung eines XML-Dokumentes einen typischen Anwendungsfall einer modernen E-Business-Plattform darstellt und sich dieser mit dem Dokumentmigrationsmodell besonders gut lösen lässt. Wie bei der Vorstellung der Prozessdiagramme (siehe Abschnitt 6.2) gezeigt wird, lässt sich das Dokumentmigrationsmodell in einer Modellierungssprache sehr einfach ausdrücken, da auf eine explizite Angabe der einzelnen Datenflüsse verzichtet werden kann und so die Diagramme wesentlich übersichtlicher und leichter verständlich werden.

Hinzu kommen die Argumente, dass das Migrationsdokument Workflow-relevante Daten laut Workflow-Referenz-Modell enthält, die sowieso jeder Aktivität mitgeteilt werden müssen, und dass durch die Festlegung auf den Austausch eines einzigen Migrationsdokumentes die Schnittstellen der Komponenten weiter vereinheitlicht werden können, da nicht die Ein- oder Ausgabe einer unterschiedlichen Anzahl von Dokumenten vorgesehen werden muss. Aus diesen Gründen favorisieren wir das Dokumentmigrationsmodell mit einem impliziten Datenfluss entlang des Kontrollflusses.

Dokumentmigration versus zentraler Datenspeicher

Um die Workflow-relevanten Daten für alle Aktivitäten zugreifbar zu machen, sind prinzipiell zwei verschiedene Ansätze denkbar. Bei dem in dieser Arbeit zu Grunde liegenden Dokumentmigrationsmodell findet, wie bereits beschrieben, ein Dokumentfluss durch die Aktivitäten des Prozesses statt, der mit dem Kontrollfluss übereinstimmt. Im Workflow-Referenz-Modell wird in diesem Zusammenhang von *direktem Datenaustausch* gesprochen. Danach werden durch geeignete Mechanismen die Daten an sich, und nicht eine Referenz darauf, an die Aktivitäten übergeben: „[..] data is physically transferred to the next activity as part of the activity navigation within the process.“ (aus [Wmc95] S.26) Nach Ausführung einer Aktion müssen die Daten wieder zurück an das WFMS übertragen werden.

Demgegenüber befinden sich im Falle eines *indirekten Datenaustauschs* die Workflow-relevanten Daten in einem gemeinsamen, zentralen Speicher, der beispielsweise durch eine Datenbank realisiert werden könnte. Jede Komponente des Prozesses erhält einen Zugriffspfad, mit dessen Hilfe sie die Daten lesen oder verändern kann. Auf diese Weise sind die für die Ausführung des Prozesses benötigten Daten an einer zentralen und von überall erreichbaren Stelle abgelegt und brauchen nicht explizit übergeben zu werden.

Die beiden Ansätze Dokumentmigration und Verwendung eines zentralen Speichers entsprechen den Konzepten *Kommunikation* und *Kooperation* zum Datenaustausch im Bereich der verteilten Systeme (siehe [Wet93] S.110f). Direkter Datenaustausch kann dabei als Kommunikation aufgefasst werden, da eine Übertragung von Daten stattfindet, während indirekter Datenaustausch eine Kooperation mittels gemeinsamem Speicher darstellt. Beide Ansätze sind laut [Hei00A] zunächst lediglich als konzeptuelle Modelle zu sehen. Bei der Realisierung können sie jedoch aufeinander abgebildet werden und sind aus diesem Grund als gleichmächtig anzusehen.

Auch das Konzept der in dieser Arbeit zu Grunde gelegten Migration eines XML-Dokuments ist eher als von der späteren Implementierung unabhängige Vorstellung zu betrachten. Es lässt sich bei der konkreten Umsetzung sowohl durch direkten wie indirekten Datenaustausch realisieren. Bei der von uns gewählten objekt-orientierten Realisierung werden bei der Ausführung von Workflow-Modellen nicht die XML-Daten selbst an eine Softwarekomponente übergeben, sondern eine Referenz auf die entsprechenden Objektinstanzen im Speicher. Über diese Referenz können die Komponenten dann intern die Daten lesen oder verändern. Daher findet kein Dokumentfluss im eigentlichen Sinne statt, sondern ein Fluss der Referenzen durch den Prozess.

6.1.2 Komponentenintegration

Eine zentrale Anforderung aus der praktischen Anwendung (vgl. Fallstudie in Abschnitt 3.1) ist die Anbindung der bestehenden Softwaresysteme und -komponenten und ihre Integration in die einzelnen Aktivitäten des Workflows. Dies ist besonders unter dem Ziel einer vollständig automatisierten Ausführung der Workflows wichtig, bei der nicht nur die Steuerung des Kontroll- und Datenflusses, sondern auch die einzelnen Aktivitäten selbst ohne Benutzerinteraktion ausgeführt werden sollen. Dies macht es nötig, bestehende Softwarekomponenten für die Ausführung der einzelnen Aktivitäten einzubinden.

Mit dem Begriff Softwarekomponente sind in dieser Arbeit die verschiedensten Software-Einheiten eines Unternehmens gemeint, die jeweils ganz unterschiedliche Funktionen wahrnehmen können. Es kann sich dabei um Module handeln, die reine Datenverwaltung in Verbindung mit Datenbanken oder anderen Speichereinheiten betreiben, Daten transformieren, Daten aus anderen Systemen anfordern oder Daten auf Basis spezieller Geschäftslogiken auswerten und entsprechende Berechnungen durchführen. In der Regel findet sich in einem Unternehmen eine ganze Landschaft heterogener Systeme von betrieblicher Standardsoftware wie SAP R/3 über umfangreiche Datenbanksysteme bis hin zu objektorientierten Anwendungssystemen auf Basis von Enterprise Java Beans. Strategisches Ziel einer prozessorientierten E-Business-Plattform ist es, möglichst alle diese Softwarekomponenten einzubinden und in die Workflow-Modelle zu integrieren.

Bei der Vorstellung der Architektur eines Internet Portal Systems schreiben Gruhn und Schöpe in [Gru01] dazu: „The business transactions conducted between two partners in electronic commerce or electronic business are supported entirely or in part by different software systems. […] In this sense, an electronic commerce portal is an integration platform for different software systems like legacy, web, Internet or office systems." (siehe [Gru01] S.2).

Bei der Lösung dieser Aufgabe stellt sich als zentrale Frage das Problem der Schnittstellen, die benötigt werden, um die verschiedenen Komponenten anzusprechen und Daten mit ihnen auszutauschen. Wünschenswert wäre es, wenn das WFMS zu allen Komponenten eine passende Schnittstelle vorfände, der es das aktuell vorhandene Migrationsdokument übergibt und dann auf die Rückgabe eines transformierten Dokumentes wartet. Da allerdings nicht davon auszugehen ist, dass alle Komponenten eine solche Schnittstelle mit Dokumentübergabe anbieten, ist es nötig, kleine Programme zu entwickeln, die eine Art Brückenfunktion zwischen dem WFMS und den einzelnen Komponenten übernehmen. Diese sogenannten *Konnektoren* bieten dem WFMS die gewünschte Schnittstelle zur Datenübertragung, müssen aber auf der Verbindungsseite zur angesprochenen Softwarekomponente jeweils die Schnittstellen dieser speziellen Komponente unterstützen.

In [Gru01] findet sich ein ähnliches Konzept unter der Bezeichnung *Adaptoren*. Dort heißt es: „Since the interfaces used to access the external systems are very different, each one is connected to the central middleware „backbone" via an individual adaptor." (siehe [Gru01] S.4).

Es ist außerdem nicht davon auszugehen, dass alle zu integrierenden Softwarekomponenten das vom WFMS intern benutzte Datenformat unterstützen. Damit ergibt sich als zweite wichtige Aufgabe für die Konnektoren eine Transformation des internen Datenformats in ein für die Softwarekomponente verständliches Format mit anschließender Rücktransformation.

Die folgende Grafik zeigt exemplarisch, wie ein SQL-Konnektor als Verbindungsbrücke zu einer Datenbank eingesetzt wird, um SQL-Befehle auszuführen, die in dem Migrationsdokument spezifiziert sind. Der Konnektor hat eine Schnittstelle zum WFMS, die das dort für die Migrationsdokumente verwendete Datenformat versteht, sowie eine Schnittstelle zum Datenbanksystem, die unter Umständen auf anderen Formaten oder Protokollen basiert.

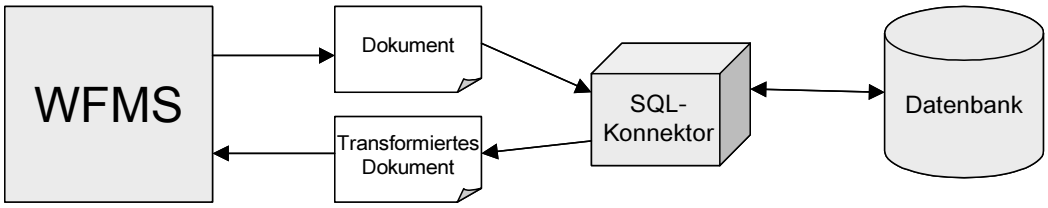


Abb. 6.2: Prinzip eines Konnektors

Da der Konnektor in diesem Beispiel speziell die Umsetzung zwischen WFMS und Datenbank vornimmt, ist er nur für die Einbindung der Datenbank geeignet. So erfüllt ein Konnektor in der Regel nur die Brückenfunktion für eine ganz bestimmte Komponente. Soll vom WFMS eine andere Komponente eingebunden werden, ist es in den meisten Fällen nötig, eine neue Konnektorklasse zu implementieren. Dabei lassen sich

spezielle Teile wie die Verarbeitung des WFMS-Datenformats durch Generalisierung und Vererbungsstrukturen wiederverwenden, so dass der Aufwand zur Erstellung des Konnektors minimiert wird. Hervorzuheben ist auch, dass ein einmal realisierter Konnektor an vielen Stellen und auch in mehreren Workflow-Modellen zur Integration seiner zugehörigen Komponente benutzt werden kann. Trotzdem ergibt sich ein nicht zu vernachlässigender Wartungsaufwand. Wird nämlich die Schnittstelle der Softwarekomponente verändert, dann muss auch der zugehörige Konnektor entsprechend angepasst werden (vgl. [Gru01] S.5).

Über das Konzept der Konnektoren lässt sich für jede Aktivität des Workflows eine bestimmte Komponente aus den existierenden Softwaresystemen einbinden. Neben den vorhandenen einzelnen Softwarekomponenten sollen aber auch bereits definierte Workflow-Modelle als Aktivitäten innerhalb eines Prozesses aufgerufen werden können. Die so entstehende *Hierarchisierung* hilft bei der Handhabbarkeit komplexer Prozesse und unterstützt bei der Wiederverwendung bestimmter Teilprozesse, die dann als *Prozess-Bausteine* in einem übergeordneten Prozess eingesetzt werden können.

6.1.3 Modell- und Ausführungsebene

Ein Workflow-Modell ist eine abstrakte Beschreibung für eine Menge von gleichartigen Workflows. Die einzelnen Workflow-Instanzen unterscheiden sich durch verschiedene Werte bei den zu Grunde liegenden Parametern und Daten. So kann etwa ein Workflow zur Überweisung eines Geldbetrages für verschiedene Konten oder verschiedene Beträge aufgerufen werden. Für diese verschiedenen Workflows wird aber immer das gleiche Modell instanziiert.

Neben der Abstraktion von den konkreten Dateninhalten wird in einem Workflow-Modell auch von konkreten Aufgabenträgern abstrahiert. Im Bereich der Geschäftsprozessmodellierung etwa wird in einem Geschäftsprozessmodell für die Erledigung eines Kreditantrages nicht der konkrete Kreditexperte Herr A eingesetzt, sondern die *Rolle* Kreditexperte, die von Herrn A erfüllt wird. Es könnte aber auch andere Personen geben, die diese Rolle erfüllen. Vorteil dieser Abstraktion ist die größere Unabhängigkeit von Veränderungen im Bereich der konkreten Personen. So bliebe das Prozessmodell unberührt, wenn Herr A das Unternehmen verließe. Jemand anderes, der die gleiche Rolle erfüllt, könnte die Erledigung der Aufgabe übernehmen. Ein ähnliches Abstraktionskonzept ist das der *Stelle*, zum Beispiel die Stelle des Abteilungsleiters, für die eine konkrete *Stellenbesetzung* in Form einer realen Person zu jedem Zeitpunkt gegeben sein sollte. Beide Konzepte werden in [Böh95] S.5 vorgestellt.

Bei der Beschreibung der Aktivitäten sollte nicht mehr explizit auf eine konkrete Person sondern auf eine Rolle oder Stelle verwiesen werden. Das Workflow-Management-System kann dann zur Laufzeit anhand des statischen Organisationsmodells entscheiden, welcher Bearbeiter auszuwählen ist (vgl. [Böh95] S.8).

Eine analoge Abstraktion kann auch bei der Modellierung der hier betrachteten Workflows zur Integration von Softwarekomponenten benutzt werden: Die Konnektoren realisieren, wie im vorausgehenden Abschnitt dargestellt, die Verbindungen zu den existierenden Softwarekomponenten. Abstrakte Repräsentanten der Konnektoren werden in einem statischen Modell aufgeführt, um dort ihre genaue Schnittstelle gegenüber dem WFMS zu spezifizieren. Zur Laufzeit kann das WFMS dann beliebige verfügbare Implementierungen für einen Konnektor einsetzen, solange die Implemen-

tierung die spezifizierte Schnittstelle erfüllt. Die Schnittstelle bezieht sich hier zum einen auf die Signatur der aufrufbaren Methoden, zum anderen auf den verwendeten Dokumenttyp für die ein- und auszugebenden Daten. Diese Spezifikation reicht aus, um das Workflow-Modell aufzustellen, auch wenn es zum Zeitpunkt der Modellierung noch keine konkrete Implementierung für einen der spezifizierten Konnektoren gibt.

Um aber eine Instanz des Workflow-Modells vom WFMS ausführen zu lassen, müssen für alle benutzten Konnektoren konkrete Implementierungen vorliegen. Das WFMS muss deshalb in der Lage sein, zur Laufzeit eine Instanz für jeden in dem Workflow-Modell verwendeten Konnektor zu erzeugen. Dazu müssen Informationen über die verfügbaren Konnektorklassen vorhanden sein, sowie eine Abbildung, die die Repräsentanten des statischen Modells auf konkrete Konnektorimplementierungen abbildet. Auf der *Ausführungsebene* werden die abstrakten Elemente der *Modellebene* also an konkrete Instanzen gebunden.

Vorteile der Abstraktion und Modellbildung sind ein besseres Verständnis der Zusammenhänge und Prozesse, was vor allem durch grafische Modelle gefördert wird. Bei genauer Spezifikation der Schnittstellen lassen sich Planungen, Simulationen und Optimierungen bereits durchführen, auch wenn die konkrete Implementierung einer Softwarekomponente oder eines bestimmten Konnektors noch nicht vorhanden ist. Ohne Tests mit den echten Komponenten durchführen zu müssen, kann man bereits anhand des Workflow-Modells Fehler und Konsistenzbrüche feststellen. Vor allem wird durch die Abstraktion aber eine hohe Wiederverwendbarkeit der Modelle gewährleistet. Ändert sich etwas an der internen Implementierung einer konkreten Komponente bzw. eines Konnektors, so müssen nicht alle Workflow-Modelle geändert werden, die diese Komponente benutzen. So lange weiterhin die spezifizierte Schnittstelle von der Komponente angeboten wird, kann die Implementierung beliebig ausgetauscht und verändert werden, denn die konkrete Komponente wird erst zur Laufzeit vom WFMS gebunden.

6.1.4 Management von XML-Prozessen

Bei den in den vorausgehenden Abschnitten vorgestellten Konzepten wurde bereits mehrfach das strukturierte Datenformat angesprochen, das intern vom WFMS für die Speicherung und Strukturierung der Migrationsdokumente verwendet werden soll. Gemäß den in Kapitel 3 formulierten Anforderungen setzen wir dazu XML ein. Des Weiteren wird XML benutzt, um die Prozessbeschreibung nach der Modellierung abzuspeichern. Dazu definieren wir in Kapitel 8 eine entsprechende XML-Sprache unter dem Namen *„Process Markup Language“* (PML). Nach der Modellierung eines XML-Prozesses wird das visuelle Modell nach PML übersetzt. Dieses Beschreibungsformat des Prozesses kann dann von einem *Prozessinterpreter* (vgl. Kapitel 10) verstanden und ausgeführt werden. Bei der Übersetzung nach PML handelt es sich nicht um die Konvertierung von einer hohen Abstraktionsebene auf ein feineres Niveau mit genauerer Semantik, denn die visuelle Sprache besitzt bereits eine eindeutige Semantik. So werden Fehler durch Konvertierung und Überwindung einer semantischen Lücke vermieden (vgl. auch Abschnitt 2.4). Die folgende Abbildung zeigt einen Überblick über die Architektur des geplanten WFMS.

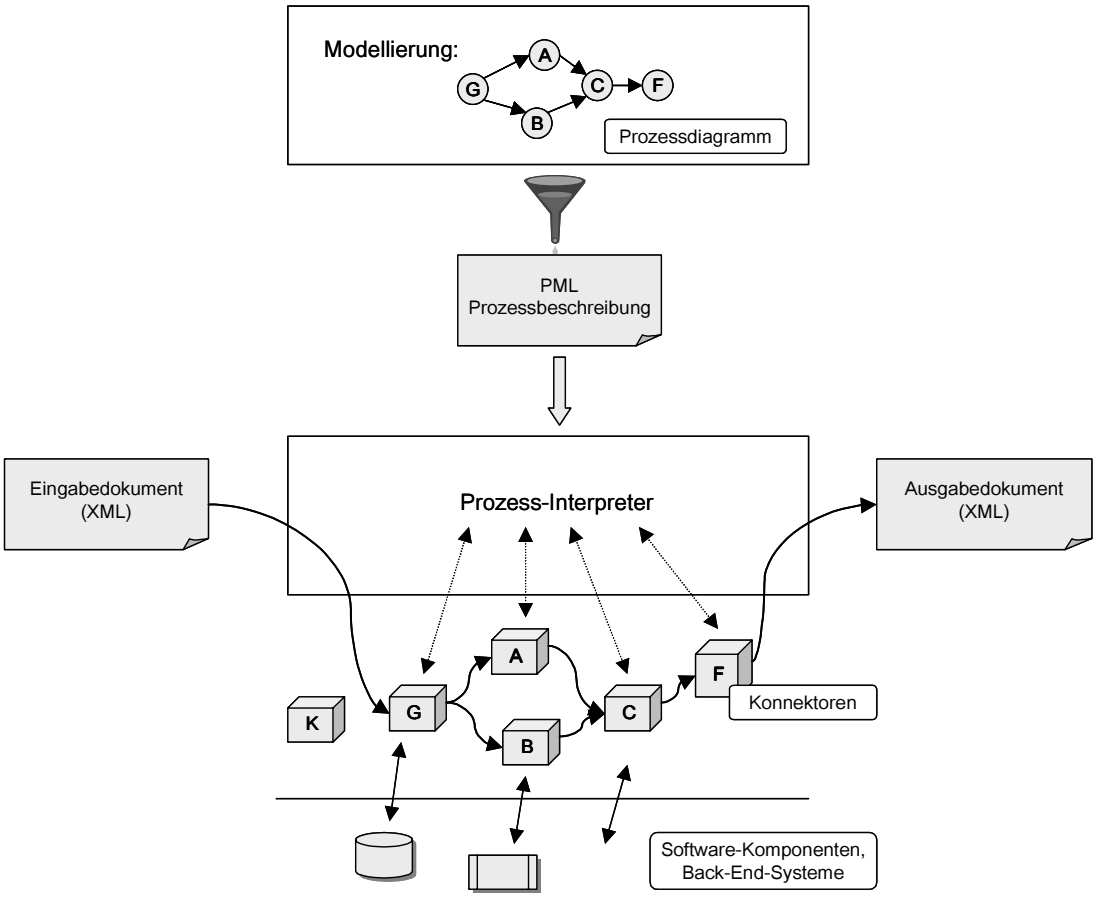


Abb. 6.3: Die Architektur des WFMS für XML-Prozesse

Der Prozessinterpreter erhält als Eingabe die Beschreibung eines Prozesses in PML sowie ein Eingabe-Dokument, auf dem eine Instanz des gewählten Prozesses ausgeführt werden soll. Durch die Speicherung der Prozessdefinition in PML ist es nicht nur möglich, die Spezifikation aus einer Datenbank oder Prozessbibliothek zu entnehmen, sondern es ist auch vorstellbar, die Prozessbeschreibung in das XML-Eingabedokument zu integrieren. Somit kann jemand, der einen ganz speziell von ihm gewünschten Prozess ausführen lassen möchte, diesen Prozess selbst entwerfen und als PML zusammen mit den Eingabedaten an den Prozessinterpreter zur Ausführung übergeben.

Der Prozessinterpreter ist das zentrale Element dieses Workflow-Management-Systems. Er muss die PML-Spezifikationen interpretieren, die einzelnen Teilprozesse anstoßen und synchronisieren, das XML-Eingabedokument entsprechend dem Prozessmodell von Komponente zu Komponente weiterleiten und so für die Integration der verschiedenen Softwarebausteine sorgen. Wie im vorausgegangenen Abschnitt vorgestellt, werden die Komponenten dabei über ihre Konnektoren angesprochen. Ziel der Arbeit ist auch die Integration des Prozessinterpreters und der zugehörigen Werkzeuge in eine E-Business-Plattform, um damit für webbasierte und andere elektronische Geschäftsabläufe ein System zu entwickeln, mit dem diese auf komfortable Art in die bestehenden Softwaresysteme integriert bzw. an diese gekoppelt werden können. Der Prozessinterpreter kann dabei als eine Ressource innerhalb der Plattform genutzt werden (vgl. Abschnitt 10.6). Nach dem Aufruf der zugehörigen Adresse zum Beispiel aus einem Portal heraus, kann mit dem Interpreter ein umfangreicher XML-Prozess

durchlaufen werden, beim dem die Daten gesammelt und aufgearbeitet werden, bevor sie an die aufrufende Einheit (z.B. den Web-Browser) zurückgegeben werden.

Für die Realisierung der hier umrissenen Konzeptideen gehen wir in den folgenden Abschnitten und Kapiteln zunächst auf die Frage nach einer geeigneten Modellierung der XML-Prozesse ein und stellen danach das System zur Ausführung der Workflows vor (Kapitel 10). Dabei sollen stets die sich aus der praktischen Anwendung ergebenen Anforderungen (vgl. Kapitel 3) an das System berücksichtigt werden.

6.2 Modellierung mit Prozessdiagrammen

Die Beschreibung von XML-Prozessen mit Prozessdiagrammen, einer Erweiterung von UML-Aktivitätendiagrammen, soll nun zunächst anhand eines Beispielprozesses erfolgen, wie er in der Praxis gegeben sein könnte. Anschließend werden allgemein die einzelnen Aspekte und Sprachkonstrukte von Prozessdiagrammen definiert. Für das Beispiel sei folgendes Anwendungsgebiet gegeben: Eine Bank bietet ihren Kunden die Möglichkeit, mit Hilfe eines Online-Banking-Systems verschiedene Kontotransaktionen im Internet auszuführen. Ein Applet beim Kunden generiert jeweils ein XML-Dokument, das die gewünschte Transaktion beschreibt und an den Webserver der Bank übermittelt. Dort soll es einen XML-Prozess durchlaufen, der die notwendigen Verarbeitungsschritte durchführt. In unserem Beispiel stehen drei Softwarekomponenten und die zugehörigen Konnektoren zur Verfügung, die als Akteure innerhalb des Prozesses dienen sollen. Dieses statische Modell des Prozesses ist in Abbildung 6.4 dargestellt.

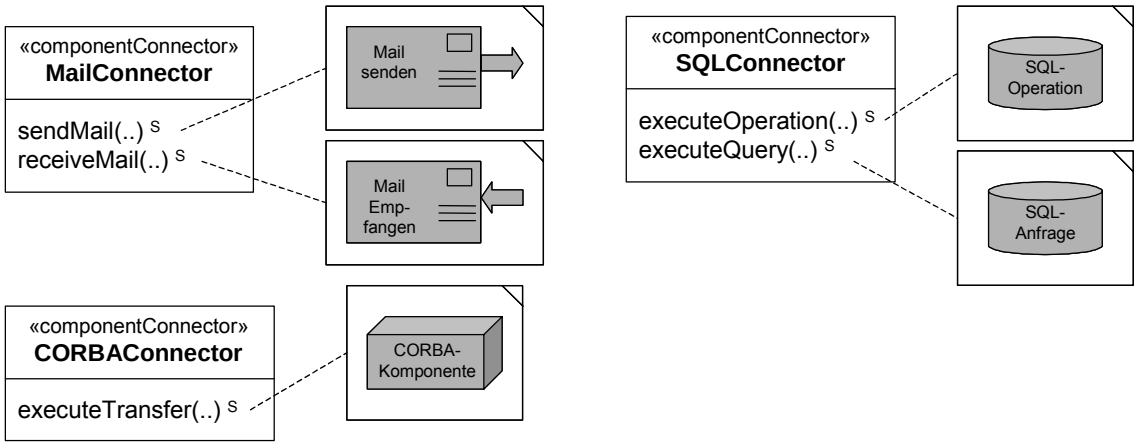


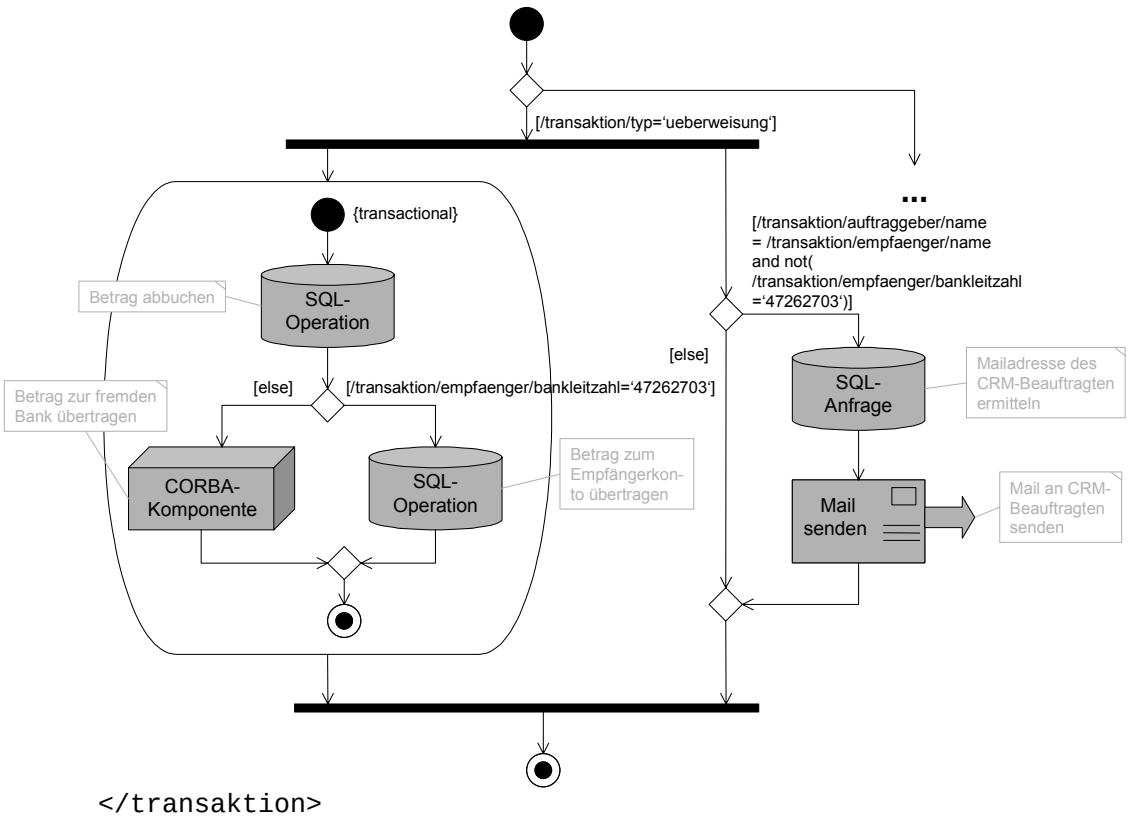
Abb. 6.4: Statisches Komponentenmodell des Prozesses

Es existiert eine Komponente, die in der Lage ist, auf eine Datenbank zuzugreifen, eine Komponente zum Ansprechen von entfernten Systemen mittels CORBA sowie eine Mail-Komponente zum Versenden und Empfangen von E-Mails. Die Komponenten werden nicht direkt in den Prozess integriert, sondern mit Hilfe von sogenannten *Konnektoren* angesprochen. Diese Konnektoren bieten dem Prozessinterpreter, der den Prozess ausführen soll, eine passende Schnittstelle zur Verarbeitung der XML-Daten. In der grafischen Darstellung jedes Konnektors werden die möglichen Operationen der zugehörigen Komponente angegeben, im Falle des SQL-Konnektors beispielsweise

„executeOperation“. Diese Operationen, die im Prozessdiagramm zur Erfüllung einer Aktivität genutzt werden können, werden im Folgenden *Services* genannt und im statischen Modell durch ein hochgestelltes „S“ gekennzeichnet. Jedem Service kann des Weiteren ein Symbol zugeordnet werden, das im Prozessdiagramm verwendet wird. Wie im Abschnitt 6.3.1 beschrieben, sollen die konkreten Implementierungen der Konnektoren erst zur Laufzeit gebunden und nicht schon im Modell festgelegt werden. Daher handelt es sich bei den im statischen Modell aufgeführten Konnektoren lediglich um Interfaces, die die Service-Operationen und ihre Parameter spezifizieren.

Abbildung 6.5 zeigt einen Ausschnitt aus dem Prozessdiagramm, der die Bearbeitung einer Überweisung beschreibt. Dabei handelt es sich um ein UML-Aktivitätendiagramm, in dem zusätzliche Konstrukte enthalten sind, die für die Beschreibung des Prozesses erforderlich sind. In einem konkreten Fall könnte der Prozess folgendes XML-Dokument von einem Kunden als Eingabe erhalten:

```
<transaktion>
      <typ>ueberweisung</typ>
      <auftraggeber>
            <konto>12345</konto>
            <name>Hans Meier</name>
      </auftraggeber>
      <empfaenger>
            <konto>67890</konto>
            <bankleitzahl>47260121</bankleitzahl>
            <name>Hans Meier</name>
      </empfaenger>
      <betrag waehrung='DM'>5000<betrag>
```




```
</transaktion>
```

Abb. 6.5: Beispiel eines Prozessdiagramms

Zunächst wird der Typ der Transaktion überprüft. Nur wenn es sich um eine Überweisung handelt, verzweigt die Ausführung in den dargestellten Teilprozess. In den Guards der Verzweigung werden XPath-Ausdrücke verwendet, um entsprechend die Daten des XML-Dokuments abzufragen. Es werden nun zwei Prozessstränge erzeugt, die parallel abgearbeitet werden.

Der linke Teilprozess ist für die eigentliche Ausführung der Überweisung zuständig. Da es sich dabei um eine atomare Folge von Aktionen handeln muss, wird mit Hilfe eines Subaktivitätszustands ein Unterprozess gebildet, welcher transaktional, d.h. entweder vollständig oder gar nicht ausgeführt wird. Diese Forderung wird durch die Angabe `{transactional}` notiert. Als erstes wird durch eine SQL-Anweisung, die von der SQL-Komponente ausgeführt wird, der Überweisungsbetrag vom Konto des Auftraggebers abgebucht. Danach unterscheidet der Prozess zwei Fälle: Falls die Bankleitzahl des Empfängers mit der Bankleitzahl dieser Bank übereinstimmt, es sich also um eine interne Überweisung handelt, kann direkt mit Hilfe einer weiteren SQL-Anweisung der Betrag dem Empfängerkonto gutgeschrieben werden. Sie wird wiederum von der SQL-Komponente ausgeführt. Falls jedoch die Überweisung zu einem fremden Geldinstitut erfolgen soll, wird mit Hilfe der CORBA-Komponente ein Auftrag an dieses Geldinstitut gesendet. Damit ist der linke Teilprozess beendet und kann mit dem parallelen rechten vereinigt werden.

Im rechten Strang erfolgt eine Prüfung der Überweisung. Falls der Name des Auftraggebers mit dem des Empfängers übereinstimmt und zudem eine unterschiedliche Bankleitzahl angegeben wurde, kann davon ausgegangen werden, dass der Kunde ein Konto bei einer anderen Bank besitzt. Dies ist bei der beispielhaften Überweisung der Fall. Daher soll die Customer-Relationship-Management-Abteilung (CRM) benachrichtigt werden. Zu diesem Zweck erfolgt zunächst eine Datenbankanfrage, die die aktuelle E-Mail-Adresse des CRM-Beauftragten ermittelt und in die XML-Daten schreibt. Anschließend wird mit Hilfe der Mail-Komponente eine entsprechende Mail an den Sachbearbeiter versendet. Dem Sachbearbeiter bleibt dann die weitere Bearbeitung überlassen. Er könnte z.B. Informationsmaterial über attraktive Anlagemöglichkeiten an den Kunden verschicken oder ein persönliches Beratungsgespräch vereinbaren.

Sobald beide Teilprozesse beendet sind, kann ein Ergebnisdokument an den Kunden ausgegeben werden, damit er über Erfolg oder Scheitern seines Auftrags informiert wird. Falls Fehler aufgetreten sind, wurden diese von den verschiedenen Komponenten in die XML-Daten geschrieben. Lag kein Fehler vor, könnte das XML-Ausgabedokument beispielsweise folgendermaßen aussehen:

```
<transaktion>
      <status>noError</status>
</transaktion>
```

In den folgenden Abschnitten sollen diejenigen Sprachelemente von Prozessdiagrammen, die eine Erweiterung von UML-Aktivitätendiagrammen darstellen, in Bezug auf ihre Semantik ausführlich betrachtet werden. Dabei entspricht jeder Abschnitt der Umsetzung einer Anforderung aus Kapitel 3.

6.2.1 Aktivitäten zur Komponentenintegration

Der Sinn jedes Aktivitätszustandes in einem Prozessdiagramm ist es, eine bestimmte Softwarekomponente in den Workflow zu integrieren. Sie ist aus Sicht des Prozessinterpreters als Black-Box anzusehen, die als Eingabe das XML-Dokument, das durch den Prozess wandert, erhält und beliebige Daten daraus lesen, Veränderungen daran vornehmen und externe Aktionen ausführen kann. Beim Beenden der Aktivität nimmt der Prozessinterpreter die möglicherweise veränderten XML-Daten wieder entgegen, bestimmt den Folgezustand und reicht das XML-Dokument nach dorthin weiter.

Es ist denkbar, dass eine Komponente und damit auch der zugehörige Konnektor nicht nur eine, sondern mehrere verschiedene Operationen bzw. Services anbietet. Jeder Service entspricht einer eigenen Methode des Konnektors. In einem Aktivitätszustand wird genau einer der möglichen Services ausgeführt. Die Semantik eines solchen Zustandes ist es also, dass zunächst eine Objektinstanz der Konnektorklasse für die gewünschte Softwarekomponente gefunden bzw. falls nötig neu erzeugt und dann darauf die passende Methode aufgerufen wird, die den Service repräsentiert. Dabei werden Werte für die vom Service geforderten Parameter übergeben.

Für jeden Service eines Konnektors kann man im statischen Modell eine grafische Darstellung festlegen, die im Diagramm anstelle des Standardsymbols für jeden Aktivitätszustand verwendet wird, in dem der jeweilige Service genutzt wird.

6.2.2 Parametrisierung von Komponenten

In vielen Fällen werden beim Aufruf eines Service Parameterwerte zu übergeben sein. Dies können im Falle einer E-Mail-Komponente Empfängeradresse, Betreffzeile und Inhalt der Nachricht oder im Falle einer Datenbank-Komponente eine SQL-Anweisung sein. Sie müssen in den Aktivitätszuständen des Diagramms angegeben werden. Im Zusammenhang mit XML-Prozessen sind drei Möglichkeiten denkbar, woher solche Angaben stammen können. Aufgrund dessen ist es notwendig, zusätzlich zum Wert eines Parameters anzugeben, auf welche Variante er sich bezieht.

Statische Werte: Zum einen ist es möglich, dass die Werte von Parametern bereits zum Zeitpunkt der Modellierung des Prozesses bekannt sind. Dabei könnte es sich z.B. um einen Server mit einer festen Adresse handeln, von dem Daten angefordert werden sollen, oder eine ganz bestimmte E-Mail-Adresse, die eine Nachricht erhalten soll. Solche Informationen können fest in der Prozessbeschreibung verankert und vom Prozessinterpreter bei der Ausführung an den Service der Komponente übergeben werden.

Werte aus dem XML-Dokument: Weiterhin können sich die benötigten Werte in dem XML-Dokument befinden, das durch den Prozess wandert. Dort könnte beispielsweise eine Liste von E-Mail-Adressen vorliegen, die alle eine Nachricht erhalten sollen. Um auf bestimmte Stellen innerhalb eines XML-Dokuments zuzugreifen, dient die Sprache XPath (siehe [XPath99]). Bei der Modellierung des Prozesses muss für jeden Parameter ein XPath-Ausdruck angegeben werden, der festlegt, wo sich zur Laufzeit die Daten befinden werden. Der Prozessinterpreter kann dann bei der Ausführung die Daten mit Hilfe eines XPath-Prozessors aus dem XML-Dokument auslesen und an die Komponente übergeben. Neben XPath sind auch andere Anfragesprachen für XML-Daten wie XQL (siehe [XQL98]) oder XQuery (siehe [XQuery01]) denkbar, die jeweils Erweite-

rungen von XPath darstellen. Im Folgenden kommt ausschließlich XPath zum Einsatz, es kann aber grundsätzlich auch jede andere Sprache verwendet werden, falls die Möglichkeiten von XPath nicht ausreichen sollten oder sich künftig eine andere Anfragesprache zum Standard entwickelt.

Werte aus der Prozessumgebung: Als dritte Möglichkeit ist es denkbar, dass den Komponenten externe Werte übergeben werden müssen, die aus dem Kontext des Prozessablaufs stammen. Dies könnte beispielsweise die aktuelle Systemzeit oder Ähnliches sein. Da XML-Prozesse im Anwendungsbereich von Web-Applikationen zu sehen sind, liegt es nah, die Technik der sogenannten *Sessions* zu benutzen. In der Spezifikation des Java Servlet Technologie (siehe [Cow01] S.52) ist eine Möglichkeit festgelegt, wie sich eine Web-Anwendung Statusinformationen zu einen bestimmten Client über mehrere Anfragen hinweg merken kann. Zu diesem Zweck können Attributwerte unter einem eindeutigen Namen an ein sogenanntes *Session-Objekt* gebunden werden, von dem es genau eines pro anfragendem Client gibt. Aufgrund des Einsatzes von XML in E-Business-Anwendungen ist es denkbar, dass in diesem Session-Objekt auch XML-Daten abgelegt werden. In sunShine (siehe Abschnitt 4.3) beispielsweise wurde dieser Ansatz dahin gehend erweitert, dass beliebig viele XML-Bäume gespeichert werden können. Jeder XML-Baum ist dabei unter einem eindeutigen Namen bekannt und wird *Sessioncontext* genannt. Zum Zugriff auf solche Daten wird bei Prozessdiagrammen ebenfalls XPath verwendet. Für jeden Parameter eines Service kann also ein XPath-Ausdruck angegeben werden. Der Prozessintepreter liest dann bei der Ausführung des Prozesses die Daten aus der entsprechenden Stelle eines XML-Baums und übergibt sie an die Komponente.

Schließlich stellt sich die Frage nach dem Datentyp eines Parameters. Eine Service-Methode könnte prinzipiell jeden beliebigen Datentyp für ihre Parameter verwenden. Die Ermittlung der Werte zur Laufzeit ergibt jedoch zumindest bei den letzten beiden der oben beschriebenen Herkunftsorte zunächst nur Werte vom Typ „String“. Dies liegt daran, dass die heute verfügbaren Werkzeuge zur XML-Verarbeitung ebenfalls nur mit dem Typ „String“ arbeiten. Bevor die so gewonnenen Werte also als Parameter an einen Service übergeben werden können, müssten sie in den gewünschten Zieldatentyp konvertiert werden. Dies ist für jeden Typ individuell zu realisieren, was dazu führt, dass stets nur eine Auswahl von Typen unterstützt werden könnte. Wir haben uns aus diesem Grund dazu entschlossen, zunächst nur Parameter vom Typ „String“ für eine Service-Methode zuzulassen. Jeder Service kann dann intern den Inhalt der Parameter in geeigneter Weise deuten und ggf. eine Konvertierung durchführen. Diese Lösung schien uns flexibler zu sein, als wenn die Typbehandlung bereits im Workflow-Management-System stattfände.

6.2.3 Transitionen als XML-Dokumentfluss

Jedes Prozessdiagramm erhält bei seiner Ausführung ein XML-Dokument als Eingabe. Dieses wird entsprechend dem Prinzip der Dokumentmigration der Reihe nach durch alle Zustände des Diagramms gereicht. Auf diese Weise entsteht ein impliziter Datenfluss des Dokuments durch den Prozess; der Datenfluss muss nicht explizit durch die dafür gebräuchlichen Konstrukte der UML-Aktivitätendiagramme wie *Objektfluss-*

zustände modelliert werden. Jede Kontrollfluss-Transition des Diagramms bedeutet vielmehr neben der Kontrollfluss-Abhängigkeit gleichzeitig einen Objektfluss, den man auch mit den Sprachelementen *Objektfluss* und *Objektfluss-Zustand* hätte modellieren können, wobei ein Objekt - das XML-Dokument - von einer Aktivität geschrieben und von der nächsten gelesen wird.

Die implizite Abbildung des Datenflusses und der Verzicht auf die Verwendung von Objektfluss-Konstrukten stellt zwar eine Modifikation der bisherigen UML-Semantik dar, führt dafür jedoch zu einer beachtenswerten Vereinfachung der Modellierung und zu einer besseren Verständlichkeit der Prozessdiagramme.

6.2.4 Guard-Bedingungen

Damit der Kontrollfluss eines Prozesses dynamisch beeinflusst werden kann, muss es möglich sein, auf das XML-Dokument oder die Prozessumgebung (siehe Abschnitt 6.2.2) zuzugreifen, um bestimmte Bedingungen zu prüfen. Dies geschieht im Zusammenhang mit Verzweigungen des Prozesses durch Guard-Bedingungen, in denen XPath-Ausdrücke angegeben werden. Ein Ausdruck wird zur Laufzeit vom Prozessinterpreter im Bezug auf das XML-Dokument oder die Prozessumgebung ausgewertet und so der nachfolgende Aktivitätszustand ermittelt.

6.2.5 Behandlung von parallelen Teilprozessen

Sowohl Aktivitäten- als auch Prozessdiagramme erlauben es, Teilabläufe zu bilden, die parallel ausgeführt werden. Bei Prozessdiagrammen ergeben sich jedoch aufgrund des impliziten Dokumentflusses Probleme: Das Dokument muss zeitgleich durch die Komponenten aller parallelen Stränge fließen. Diese können Manipulationen daran vornehmen, welche anschließend korrekt zusammengeführt werden müssen, damit wieder ein einzelnes Dokument entsteht. Dieses durchläuft dann den restlichen Prozess. Grundsätzlich sind zwei verschiedene Ansätze zur Lösung dieses Problems denkbar:

Zum einen könnten alle Teilprozesse Referenzen auf ein und dieselbe Instanz des XML-Dokuments erhalten. Da aber der Zugriff der Komponenten auf die gemeinsame Datenbasis einen *kritischen Abschnitt* darstellt, müssten die Zugriffe synchronisiert werden, damit keine inkonsistenten Sichten auf die Daten entstehen (vgl. [Bac98] S.241ff). Um dies zu leisten, müsste eine Art Sperrenmechanismus auf dem XML-Baum entwickelt werden. Dieser Ansatz erweist sich jedoch als nicht realisierbar, da mit XML-Prozessen heterogene Softwarekomponenten integriert werden sollen. Eine Komponente ist dabei für den Prozessinterpreter eine Black-Box. Er übergibt ihr als Eingabe die XML-Daten und nimmt sie anschließend wieder entgegen. Er kann jedoch keinen Einfluss darauf nehmen, wie die Komponente intern mit den Daten umgeht, insbesondere kann er nicht den Zugriff auf die Daten verzögern, um eine Synchronisation zu erreichen. Die einzige Möglichkeit bestände darin, die gesamte Ausführung jeder Komponente als kritischen Abschnitt zu betrachten und entsprechend zu schützen. Dadurch ginge aber jegliche Parallelität verloren, da die Teilstränge zwar verzahnt, ihre Aktivitätszustände aber streng sequentiell ausgeführt würden.

Die zweite Alternative besteht darin, dass jeder parallele Teilprozess eine eigene Kopie des XML-Dokuments erhält, auf der seine Komponenten Veränderungen vornehmen dürfen. Wenn die Prozesse wieder zu einem Ausführungsstrang vereinigt

werden, müssen die Kopien in geeigneter Weise zusammengesetzt werden, so dass sich wieder ein einzelnes Dokument ergibt (siehe Wen[00] S.32, 71). Dies kann jedoch problematisch werden, falls gravierende Änderungen an der Struktur des Dokuments vorgenommen wurden. Es ist beispielsweise denkbar, dass ein Service in einem Teilprozess die gesamte Struktur sowie XML-Element-Bezeichnungen des Dokuments verändert hat. Es ist dann nicht rekonstruierbar, an welcher Stelle die Änderungen der anderen Prozesse eingebaut werden müssen.

Für die Realisierung von XML-Prozessen haben wir uns für eine Variante der letztgenannten Möglichkeit entschieden. Zunächst bekommt jeder Teilprozess eine eigene Kopie des Migrationdokuments übergeben und seine Aktivitäten können beliebige Transformationen daran vornehmen. Bei der Wiedervereinigung der Stränge wird jedoch nicht versucht, die einzelnen Dokumente zusammenzuführen, sondern es werden einfach alle Dokumente als Unterelemente an ein neues XML-Wurzelelement angehängt. Dabei kann es natürlich passieren, dass bestimmte Daten doppelt in dem zusammengesetzten Dokument vorkommen. Dem Workflow-Management-System fehlen jedoch die nötigen Hintergrundinformationen, um dies automatisiert zu vermeiden. Stattdessen kann bei Bedarf hinter den Join-Zustand eine Transformation geschaltet werden, die das zusammengesetzte Dokument sozusagen bereinigt und in eine Struktur überführt, die an den Rest des Prozesses weitergeleitet wird. Dieses Vorgehen ist vergleichbar mit dem Konzept der Umlaufmappen aus dem Dokumentmigrationsmodell (siehe Abschnitt 6.1.1). Dabei könnte es ebenfalls vorkommen, dass Teile der Umlaufmappe an verschiedene Sachbearbeiter gegeben werden, die zeitgleich daran arbeiten. Die Ergebnisse werden später wieder eingesammelt und in die Mappe eingegliedert. Dabei kann der Sachbearbeiter, der für diese Zusammenführung zuständig ist, den Inhalt der Mappe anpassen, bevor er sie weiterleitet.

Eine Besonderheit im Zusammenhang mit parallelen Prozessen stellen Synchronisationszustände dar. Zunächst findet an einer solchen Prozessstelle eine zeitliche Synchronisation statt, d.h. ein Teilprozess A wartet auf einen anderen Teilprozess B. Der Sinn einer Synchronisation ist es jedoch meistens, dass A eine Information benötigt, die B erzeugt. Daher kann A erst dann weiterarbeiten, wenn B an der Synchronisationsstelle angekommen ist. Aus diesem Grund ist die Semantik von XML-Prozessen an solchen Punkten analog zu den zuvor beschriebenen Join-Zuständen definiert. Prozess B legt am Synchronisationszustand eine Kopie seines XML-Dokuments ab. Wenn A den Zustand erreicht, so wird ein neues XML-Wurzelelement erzeugt, welches sowohl das aktuelle Dokument von A selbst, als auch das von B abgelegte als Unterelemente erhält. Auf diese Weise stehen A nun alle benötigten Daten zur Verfügung. Gegebenenfalls muss durch eine anschließende Transformation das zusammengesetzte Dokument in eine Form überführt werden, die der Rest von Strang A verarbeiten kann.

6.2.6 Einschränkungen gegenüber Aktivitätendiagrammen

Neben den Erweiterungen von Aktivitätendiagrammen, die in den vorhergehenden Abschnitten beschrieben wurden, sind in Prozessdiagrammen folgende Sprachkonstrukte nicht vorgesehen:

1. **Ereignisse:** Laut UML-Spezifikation ist ein Aktivitätendiagramm eine spezielle Zustandsmaschine, in der die meisten Transitionen durch die Beendigung der vorhergehenden Aktion und nicht durch externe Ereignisse getriggert wird (siehe [Omg01] 3-157):

   > „An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states."

   Da in Prozessdiagrammen neben Pseudozuständen grundsätzlich nur Aktions- und Subaktivitätszustände erlaubt sind, erschien uns die Verwendung von Ereignissen nicht als zentrale Anforderung. Aus Zeitgründen haben wir daher darauf verzichtet, eine Ereignisbehandlung in Prozessdiagramme zu integrieren.

   Das zentrale Ziel von XML-Prozessen ist die Integration bestehender Systeme in einen Workflow-Ablauf, der Schritt für Schritt vom Workflow-Management-System abgearbeitet wird. Das einzige Bindeglied zwischen den Systemen besteht dabei aus Sicht des WFMS in dem Migrationsdokument. Falls zusätzlich dazu Ereignisse oder Nachrichten ausgetauscht werden müssen, so geschieht dies innerhalb der Systeme, jedoch transparent für das WFMS. Eine explizite Modellierung im Prozessdiagramm ist daher nicht erforderlich.

2. **Objektflusszustände:** Da XML-Prozesse so konzipiert sind, dass das Migrationsdokument die einzige gemeinsame Datenquelle aller Aktivitäten ist, die dem Workflow-Management-System bekannt ist, wurde im Abschnitt 6.1.1 der implizite Datenfluss entlang des Kontrollflusses eingeführt. Eine explizite Modellierung mit Hilfe von Objektflusszuständen ist daher in Prozessdiagrammen nicht vorgesehen.

3. **Bahnen:** Bei XML-Prozessen existiert keine Organisationshierarchie von Rollen, die für die Ausführung von Aktionen zuständig sind und für die erst zur Laufzeit konkrete Akteure gefunden werden (vgl. Abschnitt 2.3.1). Stattdessen wird für jede Aktion explizit eine passende Softwarekomponente angegeben. Aus diesem Grund erschien es uns als nicht notwendig, Bahnen zur Verdeutlichung von Zuständigkeiten in Prozessdiagrammen zu verwenden.

4. **Dynamische Zustände:** In Aktivitätendiagrammen ist es möglich, die Aktionen innerhalb von Zuständen mehrfach parallel auszuführen, wobei die Anzahl der Ausführungsinstanzen dynamisch zur Laufzeit ermittelt wird. Dies dient in der Regel dazu, Mengen von Eingabedaten möglichst effizient zu verarbeiten. Da in Prozessdiagrammen die Eingabe einer Aktion aus einem einzigen Migrationsdokument besteht, schien es nicht notwendig zu sein, dieses Sprachkonstrukt zu unterstützen. Wir haben allerdings aus Zeitgründen auf eine tiefergehende Untersuchung verzichtet; das Konzept der XML-Prozesse ließe sich bei Bedarf aber sicherlich um dynamische Zustände ergänzen.

6.3 Modellierung der Dokumenttypen

Im vorausgegangenen Abschnitt wurden die verschiedenen Konzepte zur Modellierung von Kontrollflüssen und der Integration von Softwarekomponenten in Prozessdiagrammen vorgestellt. Dabei liegt dem Kontrollfluss immer der implizite Datenfluss des Migrationsdokuments zu Grunde. Welche Struktur dieses Dokument hat, wurde bisher allerdings aus der Betrachtung weitgehend ausgeklammert. Die Modellierung der Dokumentstruktur ist für die Wohldefiniertheit des Prozessmodells sehr wichtig, wenn die Einhaltung von Konsistenzbedingungen überprüft und so der Modellierer durch Rechnerunterstützung frühzeitig auf Fehler in seinem Modell aufmerksam gemacht werden soll. In diesem Abschnitt werden daher Konzepte dargestellt, mit denen sich Informationen über den Typ und die damit verbundene Struktur des Migrationsdokuments in seinen jeweiligen Zuständen in ein Prozessdiagramm integrieren lassen.

Die Struktur der Dokumente wird durch einen *Dokumenttyp* spezifiziert. Dabei muss man zwischen der Definition des Typs und seiner Verwendung in den Modellen unterscheiden. Die Typdefinition ordnet dem Namen des Typs eine bestimmte Dokumentstruktur zu. Dadurch wird definiert, wie ein Dokument dieses Typs aufgebaut sein muss. Ist ein Dokumenttyp einmal definiert, kann er im Workflow-Modell durch Angabe seines eindeutigen Namens verwendet werden.

Zur Definition von Dokumenttypen gibt es in XML bereits etablierte Standardverfahren. Da XML das für die Migrationsdokumente zu Grunde gelegte Datenformat ist, können wir diese Techniken hier aufgreifen: Sowohl das zusammen mit XML vom W3-Konsortium vorgestellte *DTD-Konzept* (Document Type Definition, siehe [XML00] Punkt 2.8) als auch der verabschiedete Standard *XML-Schema* (siehe [XSch01]) erlauben die genaue Definition eines XML-Dokumenttyps durch Baumgrammatiken (siehe [Mak01]). Die Produktionsregeln dieser Baumgrammatiken erlauben die Ableitung hierarchisch aufgebauter XML-Dokumente. Als Nachfolgestandard des DTD-Konzepts haben XML-Schemata den Vorteil, selbst ein XML-Dokument zu sein, das mit XML-Werkzeugen verarbeitet werden kann. Außerdem erlauben sie die Benutzung einer größeren Menge von Grunddatentypen und sind mit Vererbungs- und Spezialisierungskonstrukten an objekt-orientierte Konzepte angelehnt. Daher wollen wir uns in dieser Arbeit auf die Anwendung von XML-Schemata konzentrieren. Die Validierung konkreter XML-Dokumente gegen das Schema, für das sie erstellt wurden, stellt eine Standardfunktion zahlreicher XML-Werkzeuge und -Parser dar. Das WFMS kann diese Werkzeuge nutzen, um zur Laufzeit eine Validierung der zu verarbeitenden XML-Dokumente durchzuführen.

In einem XML-Schema wird eine Menge von XML-Elementen definiert. Für unsere Betrachtung sind davon nur die sogenannten *globalen Elemente* (siehe [XSch01] Punkt 2.2.2) relevant, da nur sie als Wurzelelement eines Dokumentes benutzt werden können. Globale XML-Elemente sind durch die Angabe einer URL für ihren Namensraum und ihren Elementnamen eindeutig bestimmt (siehe [XSch01] Punkt 3). Ein Dokumenttyp kann somit durch die Angabe eines globalen XML-Elements spezifiziert werden. Alle XML-Dokumente, die jenes Element als Wurzelelement haben, sind damit von diesem Dokumenttyp. Bekannt ist diese Konvention vom Verweis auf ein globales DTD-Element im Tag `<!DOCTYPE…>` (siehe [XML00] Punkt 2.8) zu Beginn älterer XML-Dokumente, die noch anhand einer DTD validiert wurden.

Durch die Angabe eines Dokumenttyps in den Workflow-Modellen wird dieser

Typ selektiert und damit der Bereich der gültigen Dokumentinstanzen auf diejenigen eingeschränkt, die die im zugehörigen Schema definierte Struktur aufweisen. In Prozessdiagrammen ist die Einschränkung auf bestimmte Dokumenttypen an mehreren Stellen nötig: Vor allem muss für jeden Service angegeben werden, welche Dokumenttypen er als Eingabe verarbeiten kann, und welche Dokumenttypen der Service als Ausgabe an die nachfolgende Komponente übergibt. Diese Ein- und Ausgabeschnittstellen der Services werden in Prozessdiagrammen durch sogenannte *Ports* spezifiziert. Ein Port ist dabei im Wesentlichen eine Menge von Dokumenttypen. Im nachfolgenden Abschnitt wird dieses Konzept im Detail vorgestellt.

Neben der Schnittstellenspezifikation für Services im statischen Modell werden Ports im Workflow-Modell eingesetzt, um analog die Ein- und Ausgabeformate der einzelnen Aktivitäten zu spezifizieren, wenn dort ein Service ausgeführt wird. Dazu können in der Regel die Vorgaben aus dem statischen Modell übernommen werden. Transitionen verbinden die einzelnen Aktivitäten zu einem Kontrollfluss. Dadurch verknüpft eine Transition den Ausgangsport ihrer Quellaktivität mit dem Eingangsport der nachfolgenden Aktivität. Eine solche Verknüpfung, die bei der Ausführung die Migration eines XML-Dokumentes impliziert, kann natürlich nur erlaubt sein, wenn Quell- und Zielport zueinander kompatibel sind, also das Ergebnis der ersten Aktivität als Eingabe für die nachfolgende Aktivität genutzt werden kann. Führt diese Konsistenzprüfung zu einem negativen Resultat, kann die Inkompatibilität zwischen Quell- und Zielport der Transition durch eine Transformation des Dokumentes überwunden werden. Dazu müsste für die Transition eine Transformationsvorschrift angegeben werden, die ein Dokument, das zum Quellport konform ist, so umwandelt, dass es danach zum Zielport passt. Stylesheets der Transformationssprache XSLT (XML-Stylesheet Language Transformations, siehe [XSLT99]) eignen sich sehr gut für die Definition solcher Transformationen.

Um zu vermeiden, diese Transformationsvorschriften bei Transitionen mit ähnlicher Konstellation jeweils neu angeben zu müssen, wird in Abschnitt 6.3.2 das Konzept der *typisierten Transitionen* vorgestellt. Dabei soll eine größere Wiederverwendbarkeit und Konsistenz dadurch gefördert werden, dass jeder Transition ein wiederverwendbarer Typ zugewiesen wird, der für gegebenen Quell- und Zielport eine passende Transformation definiert. Abbildung 6.6 gibt einen Überblick über die Fragestellungen, die durch das Port-Konzept und die typisierten Transitionen in den folgenden Abschnitten gelöst werden sollen.

Neben der Unterstützung des Modellierers bei der Konsistenzwahrung kann eine genaue Modellierung der Dokumentstrukturen auch bei der Ausführung der Workflow-Modelle nützlich sein, weil so das WFMS jederzeit die aktuelle Struktur des Migrationsdokuments gegen den im Modell vorgegebenen Typ validieren und Abweichungen feststellen kann, die durch Fehler bei der Ausführung verursacht worden sind.

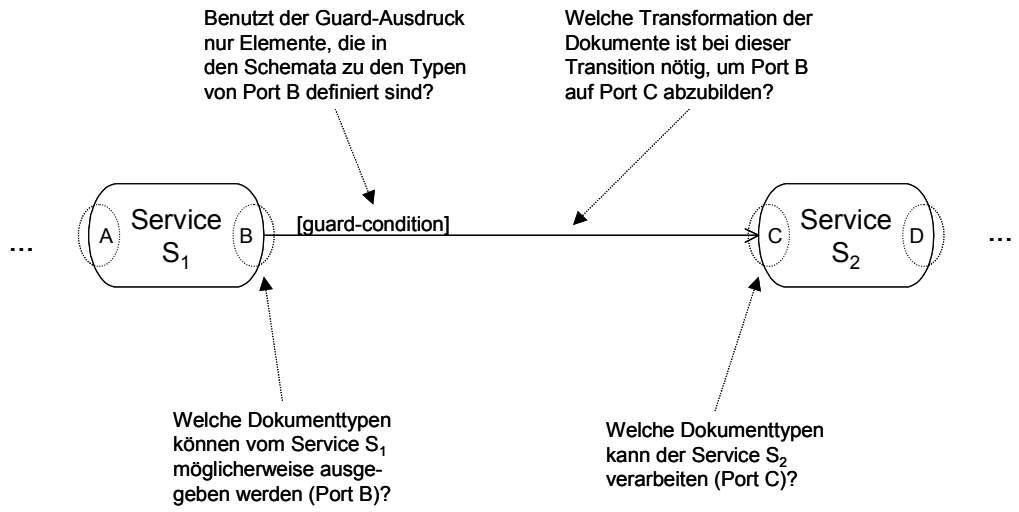

Benutzt der Guard-Ausdruck nur Elemente, die in den Schemata zu den Typen von Port B definiert sind?
Welche Transformation der Dokumente ist bei dieser Transition nötig, um Port B auf Port C abzubilden?
...
A
Service S_1
B
[guard-condition]
C
Service S_2
D
...
Welche Dokumenttypen können vom Service S_1 möglicherweise ausgegeben werden (Port B)?
Welche Dokumenttypen kann der Service S_2 verarbeiten (Port C)?

Abb.6.6: Fragestellungen bei der Modellierung der Dokumentstrukturen

6.3.1 Das Port-Konzept zur Typisierung von Dokumenten

Um ein konsistentes Prozessmodell zu erzeugen, dürfen Services nur an für sie passenden Stellen eingesetzt werden. Insbesondere können viele Services nur bestimmte Dokumenttypen verarbeiten und produzieren als Ausgabe auch nur bestimmte Dokumenttypen. Bei der Verkettung von Services in den Aktivitäten eines Prozessdiagramms müssen die Ein- und Ausgabetypen aufeinander folgender Services abgestimmt sein und zueinander passen, wenn keine Fehler entstehen sollen. Aus diesem Grund müssen Informationen über die Dokumenttypen bereits im Workflow-Modell, also im Prozessdiagramm berücksichtigt werden. Für Prozessdiagramme haben wir daher das in diesem Abschnitt vorgestellte und im übernächsten Abschnitt formal definierte Konzept der Ports eingeführt, mit dem sich Typkonsistenz bereits im Modell sicherstellen lässt.

Mit Ports lassen sich die Ein- und Ausgabeschnittstellen von Komponenten definieren, die ein XML-Dokument verarbeiten. Ein XML-Dokument hat einen bestimmten Typ, der durch sein Wurzelelement festgelegt wird. Wie die Dokumentstruktur eines gültigen XML-Elements unterhalb des Wurzelelements aussieht, bestimmen die Typdefinitionen im zugehörigen XML-Schema, das dieses XML-Element für einen bestimmten Zielnamensraum definiert. Durch Angabe dieses Namensraums und des für diesen Namensraum eindeutigen Namens des XML-Elements ergibt sich eine eindeutige Bezeichnung für das Element. Im einfachsten Fall besteht ein Port also aus der Angabe eines einzelnen XML-Elements und bestimmt damit diejenigen Dokumente, die dieses Element als Wurzelelement besitzen, als Menge der zum Port konformen XML-Dokumente.

Dieser sehr einfache Port mag für viele Services ausreichend sein, es gibt aber auch Services, die nicht nur einen, sondern mehrere verschiedene Dokumenttypen verarbeiten oder als Ergebnis produzieren können. Um auch solche Serviceschnittstellen durch Ports spezifizieren zu können, soll ein Port aus einer beliebig großen Menge von XML-Elementen bestehen. Der Port mit nur einem XML-Element ist damit nur noch ein Spezialfall des allgemeineren Konzepts.

Insbesondere kann es auch Services geben, die jedes beliebige XML-Dokument

verarbeiten können. Dies ist zum Beispiel dann der Fall, wenn der Service die Dokumentinhalte nicht direkt analysiert und verarbeitet, sondern das Dokument als Sicherungskopie in einer Datenbank archiviert. Für derartige Services kann der Port den Platzhalter #any enthalten, der ausdrückt, dass beliebige Dokumente für diese Schnittstelle erlaubt sind.

http://namespace1 elemA erlaubt alle gültigen XML-Dokumente mit Wurzelelement elemA aus dem Namensraum http://namespace1	http://namespace1 elemA, http://namespace1 elemB, http://namespace2 elemB erlaubt alle gültigen XML-Dokumente mit Wurzelelement elemA oder elemB aus http://namespace1 oder mit elemB aus http://namespace2	#any erlaubt alle gültigen XML-Dokumente mit beliebigem Wurzelelement

Tab. 6.1: Beispiele für Ports

Existenz optionaler Unterelemente

Wie bereits dargelegt wird für ein XML-Element im zugehörigen Schema definiert, welche Struktur es hat und welche Unterelemente mit welcher Kardinalität vorhanden sind. Da Schemata bewusst flexibel gehalten sind, erlauben sie auch die Definition optionaler Unterelemente sowie sogenannten *offenen* Inhalt (siehe [XSch01] Punkt 5.5), bei dem ein Teil der Substruktur unbestimmt bleibt. Um trotzdem für den Port einer Schnittstelle spezifizieren zu können, dass bestimmte Unterelemente des Wurzelelementes wirklich vorhanden sein müssen, wird die Syntax der Ports so erweitert, dass auch solche Unterelemente verlangt werden können.

Diese Erweiterung geschieht durch die Ergänzung einer Liste mit Unterelementen hinter jedem XML-Element eines Ports. Diese Liste zählt – eingerahmt von eckigen Klammern „[...]“ – alle XML-Elemente auf, die auf jeden Fall in der Baumhierarchie unterhalb des Wurzelelements vorkommen müssen. Für Elemente, deren Minimumkardinalität laut Schema sowieso größer als Null ist, ist eine solche Angabe eigentlich überflüssig, für optionale Elemente (minOccurs=0) macht sie dagegen Sinn.

Die Auflistung der Unterelemente ist besonders zur Spezifikation von Services wichtig, die das XML-Dokument nicht als Ganzes verarbeiten, sondern unabhängig von der Dokumentstruktur lediglich bestimmte XML-Elemente herausfiltern und manipulieren. Ein SQL-Service könnte zum Beispiel beliebige XML-Dokumente nach einem Element mit dem Namen <sqlStatement> durchsuchen, um die dort gekapselte Anweisung dann auszuführen. Den Eingangsport dieses Service könnte man durch den Platzhalter #any in Verbindung mit dem Subelement wie folgt spezifizieren:

```
#any [http://namespace sqlStatement]
```

Konform zu diesem Port sind somit alle XML-Dokumente, die irgendwo an beliebiger Stelle das <sqlStatement> Element enthalten, so dass der Service eine entsprechende Aktivität durchführen kann.

Notation von Ports

Zusammenfassend besteht ein Port also aus einer Menge von Dokumenttypen (die leere Menge als „∅“), wobei jeder Dokumenttyp durch die Angabe eines Wurzelelements und einer Liste obligatorischer Unterelemente (die leere Liste als „[ ]“) spezifiziert wird. Jedes XML-Element wird durch seinen Namen und einen Namensraum bestimmt. Für das Wurzelelement kann auch der Platzhalter #any verwendet werden. Somit ergibt sich unter Verwendung der erweiterten Backus-Naur-Form (EBNF, vgl. [ISO96]) die folgende Grammatik für die Notation eines Ports:

```
Port      =  (Doctype, {"," , Doctype}) | "∅" ;
Doctype   =  Root, "[", {Element}, "]" ;
Root      =  "#any"  |  Element ;
Element   =  Namespace, Name ;
Namespace =  {character} ;
Name      =  {character} ;
```

Abb.6.7: EBNF-Grammatik für Ports

Beispiele für Ports sind somit etwa:

a) http://namespace1 elemA [http://namespace1 elemB, http://namespace2 elemC], http://namespace1 elemC [ ]

b) #any [ ]

c) #any [http://namespace1 sqlStatement]

Konsistente Portverkettungen

Werden Services durch die Transitionen des Prozessdiagramms miteinander verkettet, so muss für ein konsistentes Modell gelten, dass bei jeder Transition die möglichen Ausgabedokumente des Quellservices als Eingabe für den Zielservice erlaubt sind. Dazu muss eine Bedingung formuliert werden, die sicherstellt, dass der Ausgangsport des Quellservice der Transition *kompatibel* zum Eingangsport des Zielservice ist. In der formalen Definition von Ports in Abschnitt 6.3.3 wird eine Teilmengen-Beziehung für Ports definiert, eine Berechnungsmöglichkeit gezeigt und bewiesen, dass sie als Kriterium für die Kompatibilität von Ports benutzt werden kann (siehe Abschnitt 6.3.3, Kompatibilitätssatz). Danach lassen sich zwei Ports p_1 und p_2 miteinander verketten, wenn p_1 *Teilport* von p_2 ist ($p_1 \subseteq p_2$).

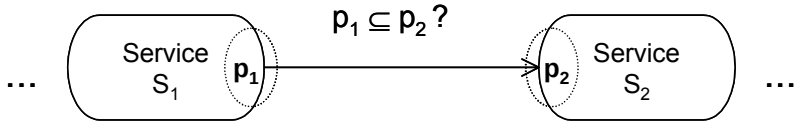


Abb. 6.8: Kompatibilitätsproblem bei Port-Schnittstellen

Um möglichst viele Services flexibel miteinander verketten zu können, ist es daher sinnvoll, die Ausgangsports der Services möglichst einzuschränken und wirklich nur die Menge der möglichen Ausgabedokumente zu typisieren. Insbesondere sollte der Platz-

halter #any bei Ausgangsports weitgehend vermieden werden. Umgekehrt sollten die Eingangsports der Services eine maximale Menge von Eingaben definieren. Gut geeignet sind Services, die im Eingangsport ein #any besitzen, denn sie können nach jedem anderen Service folgen. Um möglichst oft die Bedingung $p_1 \subseteq p_2$ erfüllen zu können, sollte p_1 also minimal, p_2 maximal gewählt werden, wobei die Gegebenheiten der Services natürlich berücksichtigt werden müssen.

Offene Ports und Bindungen

Die Überlegungen aus dem vorausgegangenen Absatz führen zu einem neuen Problem: Viele Services, deren Eingangsport mit #any gekennzeichnet ist, weil sie beliebige Dokumenttypen verarbeiten können, haben in der Regel auch einen großen Variationsbereich bei den Dokumenttypen ihrer Ausgaben. Der oben erwähnte Archivierungsservice würde etwa beliebige Dokumente in einer Datenbank als Kopie sichern und das Dokument danach an den nachfolgenden Service weitergeben. Daher müsste auch der Ausgangsport dieses Service mit #any gekennzeichnet sein, was laut vorausgegangener Betrachtung jedoch möglichst vermieden werden sollte.

Bei genauerer Analyse dieser Art von Services fällt auf, dass zwar prinzipiell ein beliebiger, unvorhersehbarer Dokumenttyp ausgegeben wird, dieser Typ aber eingeschränkt werden kann, wenn man den Service im Kontext des gesamten Prozessdiagramms, insbesondere der vorausgehenden Service-Aktivitäten betrachtet. Die folgende Abbildung zeigt noch einmal den Archiv-Service in einem Prozessdiagramm-Ausschnitt.

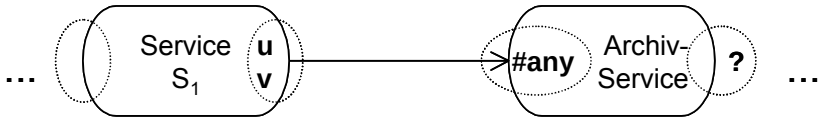


Abb. 6.9: Offener Port im Diagrammkontext

Durch die Einbettung in diesen Diagrammkontext wird klar, dass die Ausgabe des Archiv-Service nur Dokumente vom Typ u oder v sein können, die als Eingabe vom Vorgänger-Service S_1 übernommen werden. In einem anderen Kontext kann die Ausgabe vom Archiv-Service wieder ganz anders sein, so dass wir ohne den Diagrammkontext – also im statischen Modell bei der Spezifikation des Service – keine Aussage über die Dokumenttypen treffen können und deshalb eigentlich den unbestimmten Platzhalter #any angeben müssten.

Um dies zu vermeiden, benutzen wir *offene Ports*. Das sind Ports, die die Platzhalter #copyRoot und #copySubs enthalten. Diese Platzhalter werden erst dann durch konkrete XML-Elemente ersetzt, wenn der offene Port an einen anderen, nicht-offenen Port *gebunden* wird. Der Platzhalter #copyRoot kann anstelle des Wurzelelements eines Dokumenttyps angegeben werden. Bei der Bindung an einen nicht-offenen Port wird der Platzhalter durch die Wurzelelemente dieses Ports ersetzt; daher die Bezeichnung 'copyRoot'. Hinter dem Platzhalter kann wie bei gewöhnlichen Wurzelelementen auch eine Liste von Unterelementen angegeben werden, die nach der Bindung an die kopierten Wurzelelemente angehängt werden.

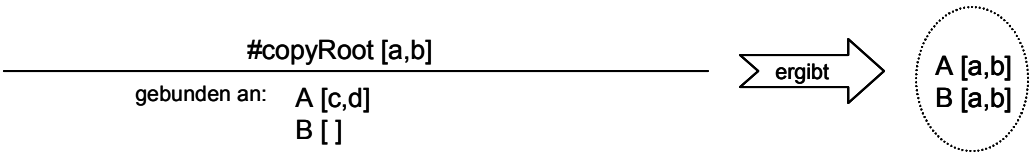


Abb. 6.10: Beispiel für die Bindung des Platzhalters #copyRoot

Durch den Platzhalter #copySubs, den man zusätzlich in der Unterelementliste zu #copyRoot angeben kann, wird außerdem das Kopieren der jeweiligen Unterelemente ausgelöst.

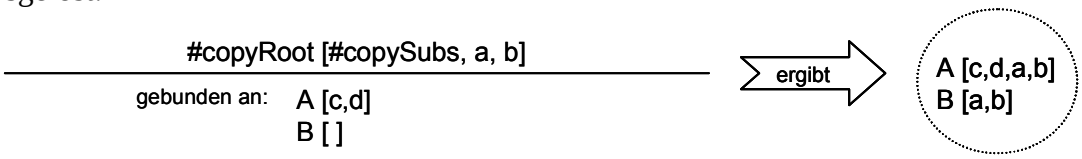


Abb. 6.11: Beispiel für die Bindung des Platzhalters #copySubs

Ein Port, der ausnahmslos die Einstellungen des Ports, an den er gebunden wird, übernehmen soll, schreibt sich demnach: '#copyRoot [#copySubs]'. Die folgende Abbildung illustriert eine Anwendung dieses Ports für den erwähnten Archiv-Service. Der offene Port wird dabei an den Ausgangsport des Vorgängerzustands gebunden.

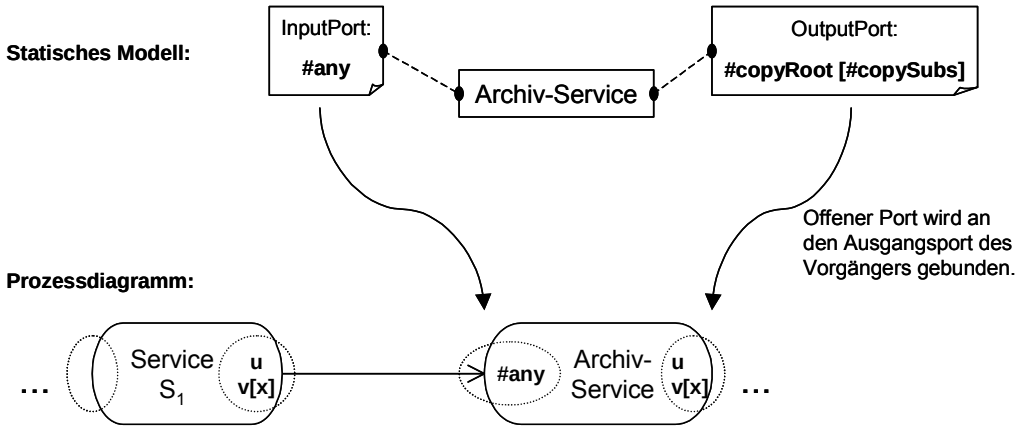


Abb. 6.12: Bindung eines offenen Ports mit #copyRoot im Diagrammkontext

Die beiden Platzhalter lassen sich jeweils um eine Menge von XML-Elementen ergänzen, die beim Kopieren ausgenommen werden sollen. Wenn wir für das bekannte Beispiel annehmen, dass alle Wurzelelemente außer B und alle Unterelemente außer c kopiert werden sollen, ergibt sich folgende Situation:

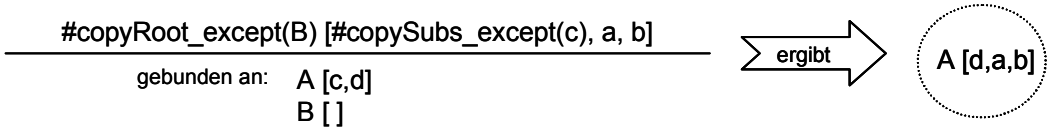


Abb. 6.13: Beispiel für die Bindung der Platzhalter mit Ausnahmen

Eine Anwendung der Ausnahmeangaben wäre zum Beispiel für den ebenfalls oben erwähnten SQL-Service denkbar. Es soll davon ausgegangen werden, dass dieser Service aus einem beliebigen Dokument mit Unterelement <sqlStatement> selbiges herausfiltert und nach Ausführung des SQL-Befehls durch <sqlResult> ersetzt. Die folgende Abbildung zeigt die zugehörigen Ports.

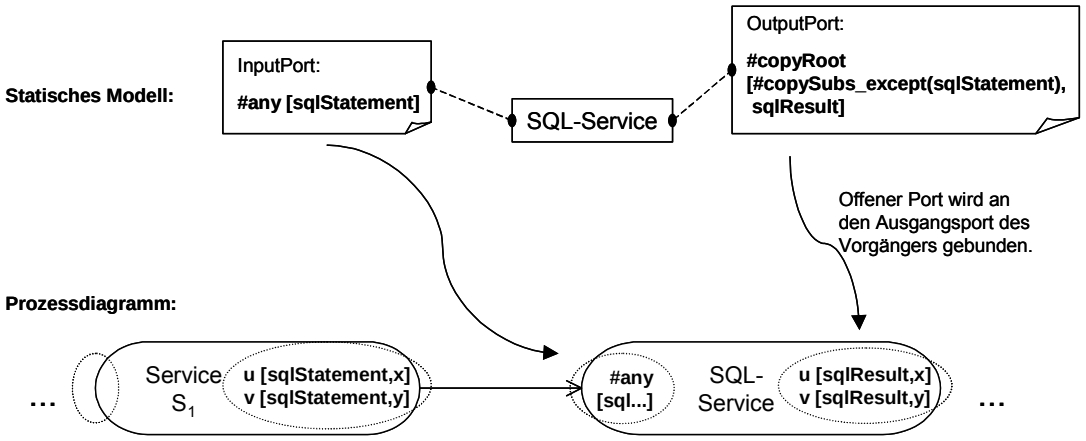


Abb. 6.14: Bindung eines offenen Ports mit ausgenommenen Unterelementen

Wie die Bindung eines offenen Ports an einen anderen Port und die dabei vorgenommene Ersetzung der Platzhalter definiert ist und berechnet wird, zeigt die formale Fundierung im übernächsten Abschnitt. Es sei darauf hingewiesen, dass jeder Port als offener Port betrachtet werden kann, denn ein offener Port muss nicht, sondern kann die Platzhalter #copyRoot und #copySubs enthalten. Umgekehrt ist jedoch nicht jeder offene Port auch ein Port, jedenfalls dann nicht, wenn er einen der Platzhalter tatsächlich enthält; erst durch die Bindung wird er zu einem nicht-offenen Port. Im formalen Modell aus Abschnitt 6.3.3 wird dies durch die Beziehung $Ports \subset OpenPorts$ ausgedrückt.

Zusammenfassend lässt sich sagen, dass offene Ports mit ihren zusätzlichen Platzhaltern #copyRoot und #copySubs die Definition offener Ausgabeschnittstellen für Services ermöglichen, deren Ausgabetypen von den Dokumenttypen abhängen, die sie von ihrem Vorgänger als Eingabe erhalten. Im Kontext eines Prozessdiagramms lässt sich der offene Port durch Bindung an den Ausgabeport des Vorgängers in einen gewöhnlichen Port umwandeln, der für die weitere Verkettung zu einem Kontrollfluss mit nachfolgenden Services besser geeignet ist, als wenn er den unbeschränkten Platzhalter #any trüge.

Notation offener Ports

Für offene Ports wird die EBNF-Grammatik von oben (vgl. Abbildung 6.7) wie folgt ergänzt, um auch die neuen Platzhalter mit den Ausnahmelisten angeben zu können:

```
OpenPort     =  (OpenDoctype, {”,” , OpenDoctype}) | “∅“ ;
OpenDoctype =  Doctype | (CopyRoot, ”[” , [OpenSubs], ”]”) ;
CopyRoot     =  “#copyRoot“  |  ( “#copyRoot_except(“, [Element, {“,“ , Element}], “)“ );
OpenSubs     =  CopySubs | ([CopySubs, “,“], Element, {“,“ , Element}) ;
CopySubs     =  “#copySubs“  |  ( “#copySubs_except(“, [Element, {“,“ , Element}], “)“ );
```

Abb. 6.15: EBNF-Grammatik für offene Ports

Die nicht-terminalen Symbole Doctype und Element werden wie bei nicht-offenen Ports abgeleitet (vgl. oben: Notation von Ports).

Entsprechende Beispiele für die durch diese Grammatik definierte Sprache wären hier:

a) #copyRoot [#copySubs, http://namespace1 elemB, http://namespace2 elemC], http://namespace1 elemC [ ]

b) #copyRoot_except(http://namespace2 elemC) [#copySubs], http://namespace1 errorMessage [ ]

Einbindung von Ports in Prozessdiagramme

Eine Einbindung von Ports als grafische Elemente in UML-Aktivitätendiagramme bzw. die darauf basierenden Prozessdiagramme ist nicht einfach, wenn gleichzeitige Einbußen bei der Übersichtlichkeit vermieden werden sollen. Grund dafür ist die doch recht technische und textlastige Notation mit Angabe von Namensräumen und Namen aller vorkommenden XML-Elemente. Vorstellbar wäre es, sie in Prozessdiagrammen als UML-Kommentare an die entsprechenden Schnittstellen der Services und Aktivitäten anzufügen. In der Regel sollten sie aber intern von einem CASE-Tool gespeichert und verwaltet werden. Der Modellierer könnte dann über Eingabedialoge des Werkzeugs die Ports für die einzelnen Schnittstellen spezifizieren.

Bleibt schließlich noch die Frage zu klären, wo überall im Prozessdiagramm und Workflow-Modell Ports spezifiziert werden müssen. Vor allem für jeden Service im statischen Modell ist jeweils ein Eingangs- und ein Ausgangsport anzugeben. Da bei einigen Services die Ausgabetypen offen sind, aber von der Eingabe abhängen, können für den Ausgangsport eines Service offene Ports benutzt werden.

Im dynamischen Teil der Workflow-Modelle hat prinzipiell jeder Zustand – also jeder Knoten des Prozessdiagramms – einen Eingangs- und einen Ausgangsport. Diese Notwendigkeit ergibt sich aus dem in Abschnitt 6.3.2 vorgestellten Konzept der typisierten Transitionen. Dabei wird nämlich vorausgesetzt, dass eine Transition stets zwei Ports miteinander verbindet. Auf den Zusammenhang von Ports und Transitionen wird dort im Detail eingegangen. An dieser Stelle wollen wir die verschiedenen Zustandstypen des Prozessdiagramms betrachten. Am wichtigsten sind dabei sicherlich die gewöhnlichen Aktivitätszustände, in denen bei Prozessdiagrammen jeweils ein Service aufgerufen wird. Für die Ein- und Ausgangsports der Aktivitätszustände werden die Ports des angesprochenen Service aus dem statischen Modell übernommen. Dabei wird

der offene Ausgangsport des Service an den gemäß dem Kontrollfluss vorausgehenden Ausgangsport gebunden. In einem vollständigen Prozessdiagramm sind daher alle offenen Ports durch Bindungen in nicht-offene Ports umgewandelt.

Dem Modellierer bleibt für einen Aktivitätszustand die Möglichkeit, die aus dem statischen Modell vorgegebenen Ports einzuschränken. Dies ist zum Beispiel sinnvoll, wenn der Eingangsport eines Service vom Wert eines Parameters abhängt. Da Parameterbelegungen im statischen Modell noch nicht bekannt sind, bleibt dort unter Umständen nur das Setzen des unbeschränkten Ports mit #any. Im Aktivitätszustand würde dieses #any zunächst aus dem statischen Modell übernommen, könnte aber vom Modellierer durch einen spezifischeren Teilport ersetzt werden, wenn er weitere Informationen über die Parameterbelegung hat. Wie in der folgenden Abbildung 6.16 illustriert, muss dabei aber immer gelten, dass der Eingangsport x'' der Aktivität Teilport des Eingangsports x des Service ist und der gebundene Ausgangsport y' des Service immer Teilport des Ausgangsports y'' der Aktivität ist. Nur dann ist ein konsistenter Dokumentfluss von x'' nach y'' möglich: Der Service definiert den Übergang von x zu y bzw. nach der Bindung des offenen Ports zu y'. Somit ergibt sich als gültiger Dokumentfluss die Portkette $x'' \subseteq x \rightarrow y' \subseteq y''$.

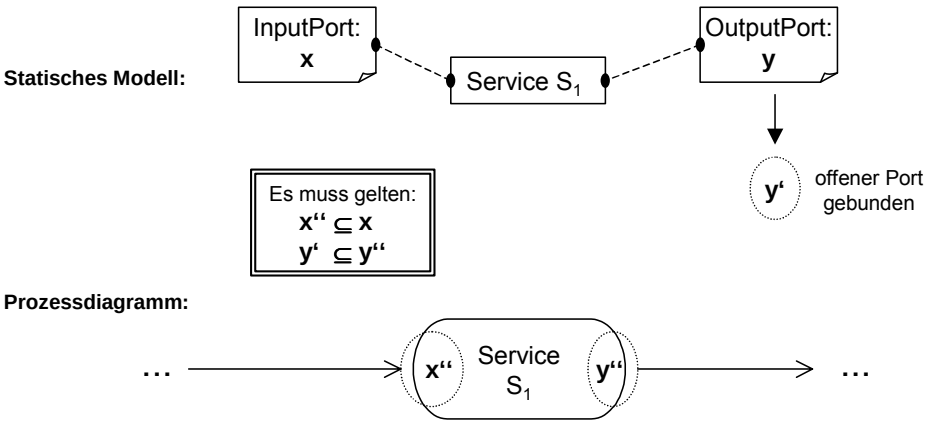


Abb. 6.16: Einschränkung von Service-Ports aus dem statischen Modell

Neben den einzelnen Zuständen muss auch für den Prozess als Ganzes ein Eingangs- und ein Ausgangsport gesetzt werden, um zu spezifizieren, welche Dokumenttypen von diesem Prozess verarbeitet und welche ausgegeben werden können. Dazu wird der Eingangsport des Prozesses vom Modellierer bestimmt und als Dokumenttyp des eingehenden Dokuments auf den Startzustand des Prozesses übertragen. Nach vollendeter Modellierung der Prozessaktivitäten auf diesem Dokumenttyp bestimmt der Ausgangsport des Endzustands den Dokumenttyp nach Durchlaufen des Prozesses. Darum wird dieser Ausgangsport des Endzustands als Ausgangsport des gesamten Prozesses übernommen. Ähnliches gilt für Subaktivitätszustände: bei ihnen müssen die Ports mit den Ein- und Ausgangsports des Prozesses übereinstimmen, der durch die Subaktivität eingebettet wird.

Die übrigen Zustände im Prozessdiagramm wie die Verknüpfungen (junction, fork, join) oder der Synchronisationszustand können alle Dokumenttypen weiterleiten und haben daher als Eingangsport den unbeschränkten Port (#any). Stärker differenzieren muss man dagegen bei den jeweiligen Ausgangports: Verzweigungen (decision und

fork) sowie der Synchronisationszustand leiten das Dokument unverändert weiter und haben als Ausgangsport daher stets den offenen Port '#copyRoot[#copySubs]', der im Kontext des konkreten Prozessdiagramms an die jeweiligen Eingangstypen gebunden wird. Nach einer Oder-Zusammenführung (merge) können Dokumenttypen aus einem der zusammengeführten Zweige vorliegen. Daher wird hier als Ausgangsport die Vereinigung aller eingehenden Ports gesetzt (vgl. Abbildung 6.17).

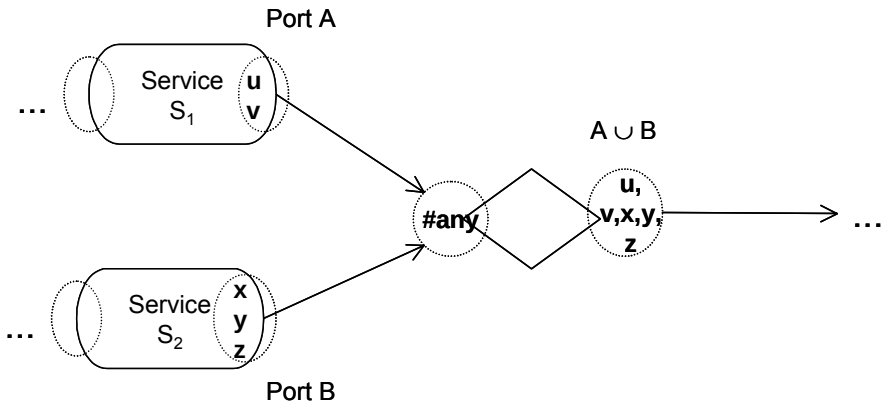


Abb. 6.17: Ports bei einer Oder-Zusammenführung (merge)

Nach einer Und-Zusammenführung (join) paralleler Stränge müssen die Migrationsdokumente aus den einzelnen Zweigen zusammengefügt werden (vgl. Abschnitt 6.2.5). Dazu wird ein neues Migrationsdokument mit dem Wurzelelement <join> erstellt, an das die einzelnen Teildokumente mit ihren Wurzelknoten angehängt werden. In der Port-Notation lässt sich das als Liste von Unterelementen darstellen, die auf jeden Fall unter dem Wurzelelement <join> zu finden sind. Wenn die eingehenden Transitionen von Ports mit mehr als einem Dokumenttyp ausgehen (wie in der folgenden Abbildung dargestellt), gibt es verschiedene Möglichkeiten, welcher Dokumenttyp nun wirklich am Ende der Transition vorliegen kann und unter das neue <join>-Element gehängt wird. Um alle Möglichkeiten zu berücksichtigen, müssen alle Dokumenttypen der eingehenden Ports wechselseitig miteinander kombiniert werden. Dies geschieht, indem das kartesische Produkt der Mengen gebildet wird. Jedes Element dieses kartesischen Produktes führt zu einem neuen Dokumenttyp im Ausgangsport des join-Zustandes mit den Einträgen des Elements als Unterelementliste unter dem Wurzelelement join (vgl. Abbildung 6.18).

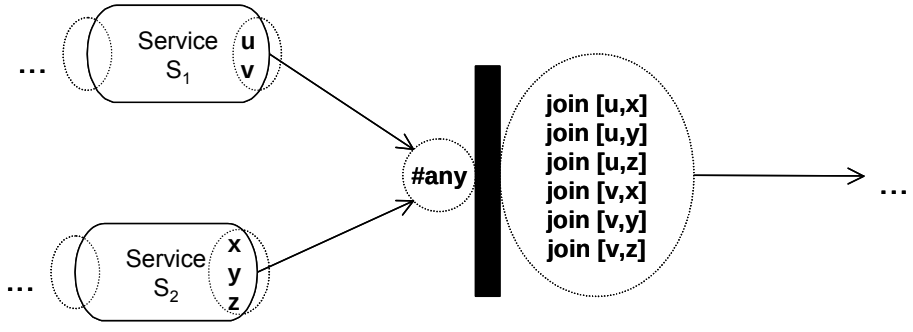
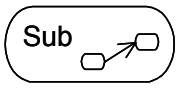
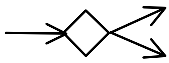
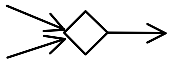
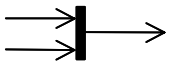
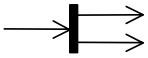


Abb. 6.18: Ports bei einer Und-Zusammenführung (join)

In der folgenden Tabelle sind noch einmal alle in einem Prozessdiagramm möglichen Arten von Zuständen zusammen mit den Ports aufgeführt, die diesen Zustandstypen im Prozessdiagramm beigeordnet werden.

	Eingangsport	Ausgangsport
Prozessdiagramm	Benutzer-definiert	= Ausgangsport des Endzustands
Service-Aktivität	Teilport des Eingangs-ports, der für den Service definiert wurde	Der gebundene offene Port des Service muss Teilport sein.
Subaktivität 	= Eingangsport des ein-gebundenen Prozesses	=Ausgangsport des einge-bundenen Prozesses
Startzustand	= Eingangsport des Prozesses	= Eingangsport des Startzustands
Endzustand	#any	Ausgangsport des Vorgängers übernehmen (#copyRoot [#copySubs])
Synchronisation	#any	Ausgangsport des Vorgängers übernehmen
Oder-Verzweigung (decision) 	#any	Ausgangsport des Vorgängers übernehmen
Oder-Zusammenführung (merge) 	#any	Vereinigung der Vorgängerports aller eingehenden Transitionen
Und-Verzweigung (fork) 	#any	Ausgangsport des Vorgängers übernehmen
Und-Zusammenführung (join) 	#any	<join> + Kartesisches Produkt über alle Vorgän-gerports für Unterelemente

Tab. 6.2: Ports bei verschiedenen Zustandsarten

Die oben dargelegten Festlegungen für die Ports der einzelnen Zustände im Prozessdiagramm werden in Abschnitt 7.3.2 als Randbedingungen der Sprachelemente von Prozessdiagrammen noch einmal formal definiert. Für das Konzept der Ports zusammen mit den typisierten Transitionen, die im folgenden Abschnitt vorgestellt werden, folgt in Abschnitt 6.3.3 eine theoretische Formalisierung.

6.3.2 Wiederverwendbarkeit und Konsistenz durch typisierte Transitionen

Bisher haben wir Ports an den Ein- und Ausgängen von Zuständen betrachtet und Bedingungen formuliert, wie eine gültige und konsistente Portverkettung zu einem Kontrollfluss aussehen muss. Die Verkettung zum Kontrollfluss geschieht in Prozessdiagrammen durch Transitionen zwischen den einzelnen Zuständen. Daher wollen wir uns in diesem Abschnitt detailliert den Transitionen zuwenden und sie im Zusammenhang mit dem Port-Konzept an unsere Anforderungen anpassen.

Transitionen mit Transformationen

Durch die oben formulierte Bedingung $p_1 \subseteq p_2$ für die erlaubte Verkettung von zwei Ports wird man bei der Wahl aufeinander folgender Services stark eingeschränkt: Als Nachfolger eines Service sind nur solche Services möglich, die die Ausgaben des Vorgängers direkt verarbeiten können. Dazu müssen die Ausgaben nicht nur den nötigen Informationsgehalt, sondern auch die passende Dokumentstruktur aufweisen, damit der nachfolgende Service weiß, an welcher Stelle er eine gesuchte Information finden kann. Nun kann es sein, dass die Ausgabe des Vorgänger-Service zwar alle nötigen Informationen enthält, diese aber nicht in der gewünschten Struktur zur Verfügung stellt. Dies zeigt sich darin, dass der Ausgangsport nicht kompatibel zum Eingangsport des Nachfolgers ist. Um dem Nachfolger aber doch die Verwendung der Daten und die Arbeit mit dem Dokument zu ermöglichen, müsste das Dokument zunächst strukturell umgeordnet werden, damit es zum Eingangsport des Service konform ist.

Um dieses Ziel zu erreichen, wird der Transition, die die beiden inkompatiblen Ports miteinander verbindet, eine Transformationsvorschrift zugeordnet, nach der die Ausgaben des Quellservice so transformiert werden, dass sie zum Eingangsport des Zielservice der Transition passen. Diese Transformationsvorschrift kann als Stylesheet in der Sprache XSLT (siehe [XSLT99]) formuliert werden.

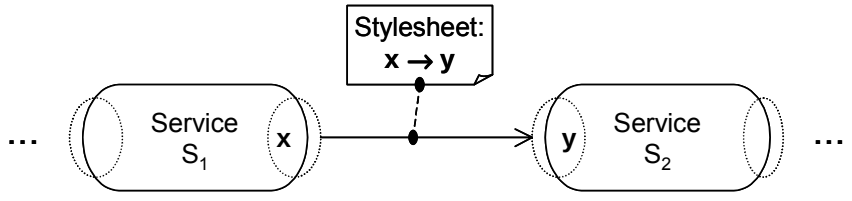


Abb. 6.19: Transition mit Transformation

Durch die Zuordnung solcher Transformationsvorschriften zu Transitionen kommen als Nachfolger einer Aktivität mit einem Ausgangsport x nicht nur Aktivitäten mit einem Eingangsport y in Frage, für den $x \subseteq y$ gilt, sondern auch solche, für die es eine Trans-

formationsvorschrift x → y gibt, die der Transition zwischen den Aktivitäten zugeordnet werden kann. Damit ist eine größere Flexibilität bei der Verkettung von Aktivitäten zu Kontrollflüssen gegeben, die umso stärker wächst, je mehr Transformationsvorschriften definiert bzw. in Form von Stylesheets angelegt werden.

Konsistenz und Wiederverwendbarkeit

Konsistenz und Wiederverwendbarkeit sind zwei sehr wichtige Ziele der Softwareentwicklung, die auch bei der Modellierung von Workflow-Modellen höchst relevant für eine fehlerfreie und effiziente Arbeit des Modellierers sind. Sie sollten daher so weit wie möglich bereits durch die hier definierten Prozessdiagramm-Sprachkonstrukte unterstützt werden.

Wiederverwendbarkeit wird vor allem durch die Elemente des statischen Modells gefördert, denn die dort definierten Services können in allen Prozessen benutzt werden. Ähnlich verhält es sich mit bereits definierten Prozessen, die als Teilprozesse in beliebigen anderen Prozessen eingebaut werden können. Im Zusammenhang mit Transitionen ist es nun wünschenswert, auch die Transformationsvorschriften und Stylesheets zur Überbrückung inkompatibler Ports möglichst oft wiederzuverwenden, insbesondere wenn gleiche Paare von Ports durch eine Transition verbunden werden sollen. Um diese Wiederverwendbarkeit zu erreichen, reicht es nicht, die Transformationsvorschriften und Stylesheets so abzulegen, dass sie zum Beispiel per Referenz auf die gleiche Quelle wiederverwendet werden können, sondern es müssen zu jedem Stylesheet auch Informationen abgelegt werden, welche Ports es miteinander verbinden kann (siehe Abbildung 6.20). Erst mit dieser Zusatzinformation ist es möglich, zu einer bestimmten Portkonstellation an einer Transition ein passendes Stylesheet aus der Sammlung bereits vorhandener Transformationsvorschriften automatisiert herauszusuchen und wiederzuverwenden.

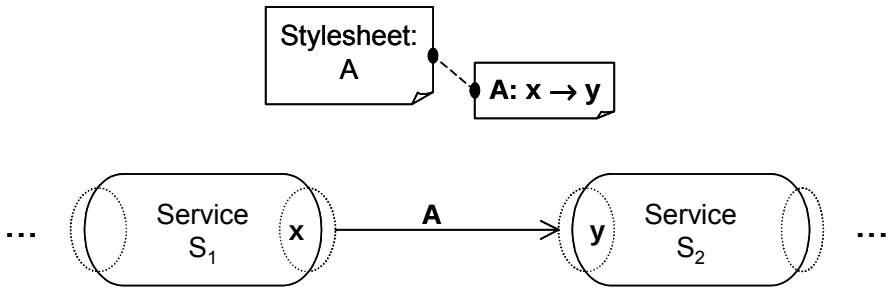


Abb. 6.20: Transformationsvorschrift mit Definitions- und Wertebereich

Zur Wahrung der Konsistenz sind wie bereits beschrieben nur bestimmte Verkettungen von Ports erlaubt. Damit wird die Menge möglicher Nachfolger einer Aktivität mit einem bestimmten Ausgangsport eingeschränkt. Um von Anfang an Konsistenzbrüche und damit fehlerhafte Modelle zu vermeiden, sollte ein Modellierungswerkzeug für Prozessdiagramme den Modellierer daran hindern, von einem Port x eine Transition zu einem Port y zu ziehen, wenn weder x ⊆ y gilt, noch ein Stylesheet vorhanden ist, das die Inkompatibilität zwischen x und y durch Transformation der Migrationsdokumente überbrücken könnte. Möchte der Modellierer trotzdem x und y durch eine Transition verbinden, müsste er zuerst ein solches Stylesheet anlegen, um die Verbindung überhaupt zu ermöglichen. Ähnlich wie bei dem Ziel der Wiederverwendbarkeit ist es somit

auch für die automatisierte Konsistenzunterstützung notwendig, dass eine Sammlung von Portpaaren jeweils zusammen mit einem Stylesheet verwaltet wird.

Typisierte Transitionen

Die angesprochene und als notwendig angesehene Sammlung von Stylesheets in Verbindung mit Portpaaren könnte Bestandteil des statischen Modells von Prozessdiagrammen werden, so dass sie an beliebigen Stellen in Prozessdiagrammen wiederverwendet werden können. Im Prozessdiagramm werden sie dann jeweils einer Transition zugeordnet, die gerade diese Ports miteinander verbinden soll. Man kann davon ausgehen, dass jede Transition mit einer solchen Transformationsvorschrift ausgestattet wird. Wenn keine Transformationen nötig sind, wird eine Identitäts-Transformation benutzt, die keine Veränderungen vornimmt. Somit kommt man schnell zu einem Typkonzept für Transitionen, bei dem die Transformationsvorschriften bzw. die gesammelten Stylesheets im statischen Modell die vorhandenen Typen für die Transitionen darstellen. Jeder Transition muss dann ein Transitionstyp – also ein Quellport, ein Zielport und eine passende Transformationsvorschrift – zugeordnet werden, was automatisch die Forderung aus der Konsistenzbetrachtung erfüllt, nach der vor dem Ziehen einer Transition erst ein entsprechender Typ angelegt werden muss, wenn es ihn noch nicht gibt. Typisierte Transitionen eignen sich somit hervorragend für die Forderung von Wiederverwendbarkeit und Konsistenz.

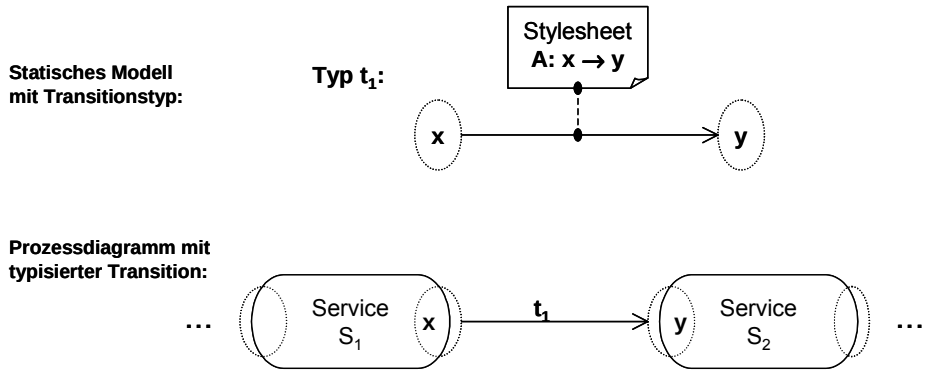


Abb. 6.21: Typisierte Transition mit ihrem Typ

Zusammenfassend besteht ein Transitionstyp also aus einem eindeutigen Namen, einem Quellport, einem Zielport und einer Transformationsvorschrift, mit der Dokumente des Quellports für den Zielport angepasst werden können. Über den Typnamen kann jeder Transition ein Transitionstyp zugeordnet werden. Die formale Definition von typisierten Transitionen folgt im nächsten Abschnitt.

Selbstverständlich muss der Typ einer Transition immer zum jeweiligen Kontext des Kontrollflusses passen; das heißt, sein Quellport muss mit dem Ausgangsport des Quellzustands der Transition und sein Zielport mit dem Eingangsport des Zielzustands der Transition übereinstimmen. Dabei wird jedoch nicht vollkommene Gleichheit, sondern lediglich Kompatibilität in Form der Teilmengenbeziehung gefordert (vgl. Abbildung 6.22).

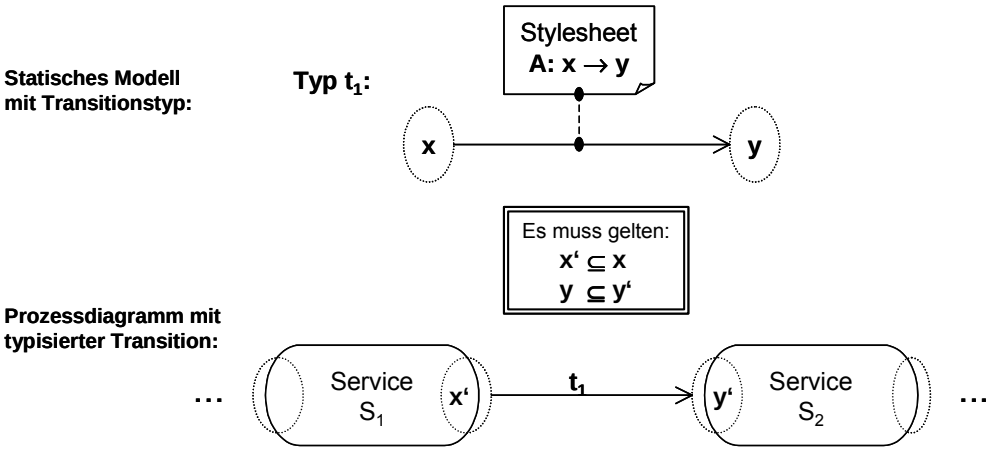


Abb. 6.22: Zusammenhang zwischen Ports der Transition und des Transitionstyps

Gelten die Beziehungen x’ ⊆ x und y ⊆ y’, so kann man sich einen gültigen Fluss des Migrationsdokuments von x’ nach y’ über die Portkette x’⊆ x → y ⊆ y’ denken, was eine wohldefinierte Transition darstellt. Dies wird innerhalb des formalen Modells aus Abschnitt 6.3.3 durch den Migrationssatz (6.18) allgemein bewiesen.

Offene Ports bei Transitionstypen

Das Stylesheet eines Transitionstyps wird während der Ausführung des Workflows von einem Stylesheet-Prozessor auf das Migrationsdokument angewandt. Insofern könnte dieser Schritt wie ein Service ebenfalls als Aktivität angesehen werden, die das Migrationsdokument manipuliert. Darum gelten für die Quell- und Zielports von Transitionstypen ähnliche Überlegungen wie für die Ein- und Ausgangsports von Services (vgl. Abschnitt 6.3.1). Insbesondere können Stylesheets sehr allgemein gestaltet sein, so dass sie auf beliebige XML-Dokumenttypen anwendbar sind oder nur bestimmte Unterelemente in der Eingabe erfordern. Der Quellport des Transitionstyps müsste in einem solchen Fall wohl mit einem #any-Platzhalter gekennzeichnet werden. Da ähnlich wie bei den Services der Ausgangsport möglichst klein sein und ein #any dort vermieden werden sollte, können auch für Ausgangsports von Transitionstypen offene Ports mit den Platzhaltern #copyRoot und #copySubs benutzt werden. Diese werden dann im Kontext des Prozessdiagramms an den Ausgangsport des vorausgehenden Zustands gebunden. Abbildung 6.23 verdeutlicht diese Zusammenhänge noch einmal. Sie zeigt ein einfaches, gültiges Prozessdiagramm mit den Definitionen der benutzten Komponenten im statischen Modell und schematisiert alle vorkommenden Ports. Die gestrichelten Pfeile zeigen, an welchen Port ein offener Port jeweils gebunden wird. Die Bindung an den Vorgänger kann sich über mehrere Zustände rückwärts fortsetzen. Spätestens der Startzustand des Prozesses hat jedoch einen Ausgangsport, der nicht offen ist und bei dem die Bindungen verankert werden können. Wie in dem Kästchen angegeben, bilden die Teilport-Beziehungen und die durch das statische Modell per Stylesheet oder Service definierten Übergänge zusammen eine konsistente Migrationskette vom Start- zum Endzustand.

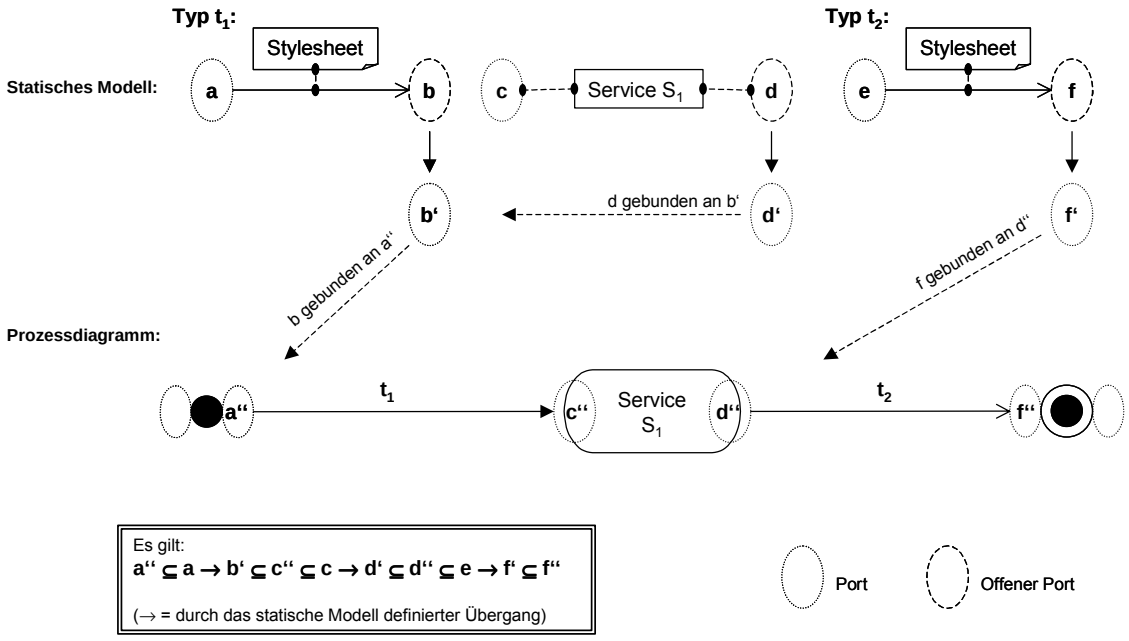


Abb. 6.23: Bindung offener Ports im Prozessdiagramm

Die Identitäts-Transition

Standardmäßig sei ein Transitionstyp vorgegeben, der benutzt werden kann, wenn Quell- und Zielport der Transition zueinander kompatibel sind und keine Anpassungstransformation durchgeführt werden muss. Solche Transitionen werden *Identitäts-Transition* genannt und mit dem Schlüsselwort #idle (engl.: leerlaufend, untätig) gekennzeichnet, weil sie das Migrationsdokument ohne jede Veränderungen an den Zielport weiterreichen. Der Identitätstyp hat als Eingangsport den Platzhalter #any, als Ausgangsport die Platzhalter #copyRoot und #copySubs (offener Port), außerdem kein Stylesheet bzw. als Transformationsfunktion die mathematische Identitätsfunktion, die jedes Dokument auf sich selbst abbildet.

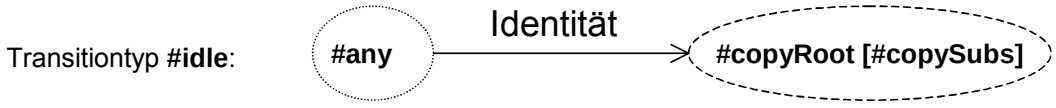


Abb. 6.24: Der Typ der Identitäts-Transition

Einbindung in Prozessdiagramme

Wie schon erwähnt stellen die Transitionstypen neben den Services und Konnektoren einen weiteren Bestandteil des statischen Modells dar. Über einen eindeutigen Namen, der zudem durch eine Package-Hierarchie gegliedert sein kann, werden die Typen den Transitions-Instanzen zugewiesen. Von zentraler Bedeutung sowohl bei der Verwaltung der Transitionstypen als auch bei der Nutzung der typisierten Transitionen sind die Funktionalitäten des Modellierungswerkzeugs. Es muss den Benutzer mit Hilfe der Transitionstypen bei der Konsistenzwahrung in seinen Modellen unterstützen. Denkbar wäre etwa eine sukzessive Entwicklung der Prozessdiagramme ausgehend vom Start-

zustand. Hat der Modellierer den Port des Anfangszustands angegeben, kann das Modellierungswerkzeug aus seinen Bibliotheken des statischen Modells diejenigen Services als Nachfolger vorschlagen, für die ein passender Transitionstyp zur Verkettung mit der zuletzt angelegten Service-Aktivität vorhanden ist. So wird der Benutzer bei der Auswahl von Services unterstützt, die nicht zu Konsistenzbrüchen führen. Vergleiche in diesem Zusammenhang auch Kapitel 9 über das im Rahmen dieser Arbeit entwickelte Modellierungswerkzeug für Prozessdiagramme.

Schließlich kann es vorkommen, dass der Modellierer zwei Ports miteinander verketten möchte, für die noch kein Transitionstyp definiert worden ist. Dann muss er zunächst einen entsprechenden Typ anlegen und ein Stylesheet entwickeln, das die nötigen Transformationen durchführt. Erst danach kann er mit der Modellierung des Workflows fortfahren und den neuen Transitionstyp benutzen. Man spricht in diesem Zusammenhang von einem *iterativen* Vorgehen, weil sich der Zyklus von Typdefinition und Typverwendung ständig wiederholt.

6.3.3 Formale Definition von Ports und typisierten Transitionen

Während in den vorausgegangenen Abschnitten relativ informell in das Konzept der Ports eingeführt wurde, erfolgt in diesem Abschnitt daran anknüpfend eine formale Definition der einzelnen Konstrukte. Dadurch wird ihnen eine präzise und eindeutige Semantik verliehen, die den Einsatz der Ports bei der Modellierung von Dokumentstrukturen im Prozessdiagramm auf eine gesicherte Fundierung stellt. Die formale Theorie hilft nicht nur, den Zusammenhang zwischen der vorab eingeführten Syntax und der nun formalisierten Semantik zu präzisieren, sondern sie dient auch als Grundlage für eine spätere Implementierung des Konzepts und begründet durch die mathematische Analyse der Zusammenhänge Berechnungsmöglichkeiten und erste Algorithmen. Außerdem lassen sich innerhalb der Theorie Zusammenhänge mathematisch korrekt beweisen, die die Konsistenz innerhalb des Port-Konzeptes sicherstellen.

Nach Abschnitt 6.3.1 soll ein Port benutzt werden, um die Schnittstelle eines Service, eines Zustands oder eines ganzen Prozesses zu definieren. Dazu ist es nötig, dass er eine bestimmte Menge von XML-Dokumenten genau spezifiziert. Um dieses Ziel auch in der formalen Semantik zu erreichen, wird zunächst dargelegt, was wir innerhalb dieser Theorie unter einem XML-Element und einem XML-Dokument verstehen wollen. Darauf basierend definieren Dokumenttypen eine bestimmte Menge von XML-Dokumenten. Eine Menge von Dokumenttypen wiederum spezifiziert einen Port.

Jeder Port kann mit den zugehörigen Dokumenttypen die zulässigen Eingaben eines Modellelements oder dessen mögliche Ausgaben definieren. Außerdem muss es möglich sein, einen Port auf Kompatibilität gegenüber einem anderen Port zu prüfen. Damit lässt sich feststellen, ob die Ausgabe des einen Elements Eingabe des nachfolgenden Elements sein kann und sie daher problemlos als Nachfolger innerhalb eines Workflows angeordnet werden können. Für diese Kompatibilitätsprüfung wird eine spezielle *Teilmengen-Beziehung* auf Ports definiert. Der daraufhin vorgestellte *Kompatibilitäts-Satz* beweist, das diese Teilmengen-Beziehung tatsächlich als Kriterium für eine konsistente Verkettung von Ports herangezogen werden kann.

Schließlich muss auch die theoretische Fundierung des Port-Konzepts genügend Flexibilität bieten, um damit Schnittstellen definieren zu können, die nur teilweise

bekannt sind, oder *offene Schnittstellen*, die erst im Kontext eines konkreten Prozessdiagramms vollständig gebunden werden können. Der zugehörige Teil dieses Abschnitts definiert, wie eine solche Bindung exakt vorgenommen werden muss, und liefert damit eine wichtige Grundlage für die spätere Implementierung des Port-Konzepts. Zum Schluss des Abschnitts werden auch die typisierten Transitionen in Zusammenhang mit ihren Ports formal definiert. Der Beweis des *Migrationssatzes* zeigt abschließend, dass durch die typisierten Transitionen konsistente Übergänge zwischen den einzelnen Diagrammzuständen vorgenommen werden können.

Die folgenden Betrachtungen fußen im Wesentlichen auf elementarer Mengenlehre und versuchen, über die Definition verschiedener Mengen der in den vorausgehenden Abschnitten vorgestellten Syntax für die Notation von Ports eine eindeutige Semantik beizuordnen. Die Absätze, die sich explizit mit den Zusammenhängen zur Syntax beschäftigen, beziehen sich dabei auf die beiden EBNF-Grammatiken aus Abschnitt 6.3.1 (vgl. Abbildungen 6.7 und 6.15).

Def. 6.1: Die Menge aller XML-Elemente

XmlElements := { (ns,s) | ∃ XML-Schema, in dem ein Element mit dem Namen s für den Namensraum ns (namespace) global deklariert wird. }

Die Menge XmlElements enthält somit alle durch irgendein Schema definierten XML-Elemente. Jedes Element wird durch seinen Namensraum und seinen Namen eindeutig identifiziert. Das Element muss global definiert sein, weil nur für globale Elemente die Eindeutigkeit des Elementnamens gewährleistet ist (siehe [XSch01] Abschnitt 3.3).

Syntax: Ein Element aus XmlElements wird durch Ableitung des Nicht-Terminals ‚Element' der EBNF-Grammatik notiert.

Def. 6.2: Die Menge aller XML-Dokumente

XmlDocs := { doc | doc ist ein wohlgeformtes und gültiges XML-Dokument }

Die Menge XmlDocs enthält somit alle XML-Dokumente, die entsprechend [XML00] Abschnitt 2.1 wohlgeformt und nach [XML00] Abschnitt 2.8 gültig bezüglich ihrer Typdeklarationen (DTD, XML-Schemata) sind. Außerdem sind für Elemente dieser Menge folgende Funktionen definiert:

r: XmlDocs → XmlElements
∀ doc ∈ XmlDocs: r(doc) = w ⇔ w ist Wurzelelement von doc

c: XmlDocs → P(XmlElements)
∀ doc ∈ XmlDocs: c(doc) = { x ∈ XmlElements | x ist in doc enthalten. }

Die Funktion r (root) liefert also das Wurzelelement der Baumstruktur zurück, die sich aus einem XML-Dokument ergibt. Die Funktion c (content) liefert alle XML-Elemente, die in dem Dokument enthalten sind.

Def. 6.3: Der #any - Platzhalter

$$\text{AnyPlaceholder} := \{ \#any \}$$

Die Menge AnyPlaceholder enthält als einziges Element das Schlüsselwort #any zur Repräsentation eines beliebigen XML-Elements. Bei der folgenden Definition der Dokumenttypen kann dieser Platzhalter anstelle eines Wurzelelements benutzt werden.

Syntax: Der #any-Platzhalter wird in der EBNF-Grammatik durch das Terminal-Symbol ‚#any' repräsentiert.

Def. 6.4: Die Menge der Dokumenttypen

$$\begin{aligned}\text{DocTypes} &:= \{ (root, subs) \mid root \in \text{XmlElements} \cup \text{AnyPlaceholder}, \\ &\qquad subs \subseteq \text{XmlElements} \} \\ &= (\text{XmlElements} \cup \text{AnyPlaceholder}) \times \mathcal{P}(\text{XmlElements})\end{aligned}$$

Ein Element aus der Menge DocTypes beschreibt den Typ eines XML-Dokumentes durch Angabe des benötigten Wurzelelementes und obligatorisch enthaltener Unterelemente. Der Platzhalter #any aus der Menge AnyPlaceholder kann alternativ zum Wurzelelement gesetzt werden und steht für ein beliebiges XML-Element.

Syntax: Ein Element aus DocTypes wird durch Ableitung des Nicht-Terminals ‚Doctype' der EBNF-Grammatik notiert. Sei dazu ein beliebiger Dokumenttyp type gegeben, mit type = (root, subs) $\in$ DocTypes und subs = $\{s_1, ...s_n\}$. Dann schreibt man den Ableitungsregeln der Grammatik entsprechend: „root $[s_1,...s_n]$".

Beispiele: #any [http://namespace1 elementA, http://namespace2 elementB]
http://namespace3 elementC []
http://namespace3 elementC [http://namespace2 elementB]

Die zu einem Dokumenttyp konformen Dokument-Instanzen sind definiert durch:

Def. 6.5: Dokumentmenge eines Dokumentsyps

$$\text{L: DocTypes} \rightarrow \mathcal{P}(\text{XmlDocs})$$

$$\forall\ type=(root, subs) \in \text{DocTypes}:$$

$$\text{L}(type) = \begin{cases} \bigcup_{x \in \text{XmlElements}} \text{L}((x, subs)) & \text{falls } root \in \text{AnyPlaceholder} \\ \{doc \in \text{XMLDocs} \mid r(doc) = root \wedge subs \subseteq c(doc)\} & \text{sonst} \end{cases}$$

L(type) heißt *Dokumentmenge vom Typ type*. Die Funktion L liefert also alle Dokumente, die einer Typspezifikation aus DocTypes genügen. Dazu muss ein Dokument das angegebene Wurzelelement besitzen und alle in subs angegebenen Unterelemente enthalten. Ist subs=$\emptyset$, entspricht L(type) der Menge aller gültigen XML-Dokumente mit dem Wurzelelement root. Ist root $\in$ AnyPlaceholder (bzw. root = #any), kann die

Dokumentinstanz ein beliebiges XML-Element als Wurzel besitzen. Daraus ergibt sich der folgende Hilfssatz, der später beim Beweis des Kompatibilitäts-Satzes benutzt wird:

Lemma 6.6:

$$\forall\, doc \in XmlDocs,\ \forall\, type=(root,subs) \in DocTypes:$$
$$doc \in L(type) \Leftrightarrow ((subs \subseteq c(doc)) \wedge (root = r(doc) \vee root \in AnyPlaceholder))$$

Beweis:

1.Fall: $root \in AnyPlaceholder$

$$\begin{aligned} & doc \in L(type) \\ \Leftrightarrow\ & doc \in \bigcup_{x \in XmlElements} L(x,subs) \\ \Leftrightarrow\ & \exists\, x \in XmlElements: doc \in L(x,subs) \\ \Leftrightarrow\ & r(doc) = x \ \wedge\ subs \subseteq c(doc) \end{aligned}$$

2.Fall: $root \notin AnyPlaceholder$

$$\begin{aligned} & doc \in L(type) \\ \Leftrightarrow\ & r(doc) = root \ \wedge\ subs \subseteq c(doc) \end{aligned}$$

Mit den bisherigen Definitionen von einzelnen XML-Elementen bis hin zu Dokumenttypen steht uns nun das nötige Rüstzeug zur Verfügung, um in der folgenden Definition zu bestimmen, was wir unter einem Port bzw. der Menge aller Ports verstehen wollen:

Def. 6.7: Die Menge aller Ports

$$Ports := \{ types \mid types \subseteq DocTypes \} = P(DocTypes)$$

Ein Port $p \in Ports$ ist somit einfach eine Teilmenge der Menge DocTypes, also eine Menge von Dokumenttypen.

Syntax: Ein Element aus Ports wird durch Ableitung des Nicht-Terminals ‚Port' der EBNF-Grammatik notiert. Das entspricht im Wesentlichen einer durch Kommata getrennten Auflistung der enthaltenen Dokumenttypen.

Die Menge der zu einem Port konformen XML-Dokumente wird wiederum durch eine Funktion L als Vereinigung über alle enthaltenen Dokumenttypen definiert:

Def. 6.8: Dokumentmenge eines Ports

$$L: Ports \rightarrow P(XmlDocs)$$

$$\forall p \in Ports : L(p) = \bigcup_{t \in p} L(t)$$

L(p) heißt *Dokumentmenge vom Port p*.

Nachdem wir nun Ports und ihre grundlegenden Eigenschaften definiert haben, fehlt noch die in Abschnitt 6.3.1 geforderte Teilmengen-Beziehung, mit der sich konsistente Portverkettungen überprüfen lassen. Für einen Port wird die spezielle Teilmengen-Relation „$\subseteq$“ wie folgt definiert:

Def. 6.9: Teilmengen-Relation für Ports

$$\forall p_1,p_2 \in \text{Ports}: \quad p_1 \subseteq p_2$$
$$\Leftrightarrow$$
$$\forall\, t_1=(root_1, subs_1) \in p_1\ \exists\, t_2=(root_2, subs_2) \in p_2:$$
$$(root_2=root_1 \vee root_2 \in \text{AnyPlaceholder}) \wedge (subs_2 \subseteq subs_1)$$

Ein Port heißt also *Teilport* eines anderen Port, wenn es zu jedem Dokumenttyp einen Dokumenttyp im anderen Port mit gleichem Wurzelelement (oder dem Platzhalter #any) und einer Menge von Unterelementen gibt, die alle auch in der eigenen Menge von Unterelementen enthalten sind.

Beispiele:			
	A[p], B[p,q]	$\subseteq$	#any[p]
	A[], B[q]	$\subseteq$	A[], B[]
	A[p], B[p,q]	$\not\subset$	#any[p,q]
	A[], B[q]	$\not\subset$	A[]

Diese Port-Relation soll benutzt werden, um festzustellen, ob zwei Ports zueinander kompatibel sind, das heißt, ob jedes Dokument, das zur Dokumentmenge des einen Ports gehört, auch zur Dokumentmenge des anderen Ports gehört. Dass dies immer gegeben ist, wenn eine Teilport-Beziehung vorliegt, zeigt der folgende Satz. Durch ihn wird die Teilmengen-Relation für Ports zur zentralen Funktion, um zwei Ports auf Kompatibilität zu prüfen. Für je zwei Ports gilt also der folgende *Kompatibilitäts-Satz*:

Satz 6.10: Kompatibilitäts-Satz

$$\forall\, p_1,p_2 \in \text{Ports}: \quad p_1 \subseteq p_2 \ \Rightarrow\ L(p_1) \subseteq L(p_2)$$

Beweis:

Seien $p_1,p_2 \in$ Ports mit $p_1 \subseteq p_2$ und $doc \in L(p_1)$ beliebig. Z.z.: $doc \in L(p_2)$

	$doc \in L(p_1)$	
$\Leftrightarrow$	$\exists\, t_1=(root_1, subs_1) \in p_1$ mit $doc \in L(t_1)$	(6.8)
$\Leftrightarrow *$	$((subs_1 \subseteq c(doc)) \wedge (root_1 = r(doc) \vee root_1 \in \text{AnyPlaceholder}))$	(6.6)

Aus $p_1 \subseteq p_2$ folgt:

	$\exists\, t_2=(root_2, subs_2) \in p_2$ mit	(6.9)
	$(root_2 = root_1 \vee root_2 \in \text{AnyPlaceholder}) \wedge subs_2 \subseteq subs_1$	
$\Rightarrow$	$(root_2 = r(doc) \vee root_2 \in \text{AnyPlaceholder}) \wedge subs_2 \subseteq subs_1 \subseteq c(doc)$	$(*)$
$\Rightarrow$	$doc \in L(t_2)$	(6.5)
$\Rightarrow$	$doc \in L(p_2)$	(6.8)

Dieser Kompatibilitäts-Satz besagt, dass wenn ein Port p_1 Teilport von p_2 ist, dann sind alle zu p_1 konformen XML-Dokumente auch zu p_2 konform. Damit haben wir eine einfache Methode gefunden, die Kompatibilität zweier Ports zu berechnen. Denn gilt die Bedingung $L(p_1) \subseteq L(p_2)$, dann kann eine Komponente mit p_1 als Ausgangsport mit einer Komponente verknüpft werden, deren Eingang durch p_2 spezifiziert ist.

Offene Ports und ihre Bindung

Im Folgenden sollen auch die in Abschnitt 6.3.1 vorgestellten offenen Ports formal definiert werden. Aufbauend auf den bereits vorgenommenen Definitionen ergibt sich daraus eine formale Semantik für die Sprache der zweiten EBNF-Grammatik aus dem erwähnten Abschnitt (vgl. Abbildung 6.15), mit der sich offene Ports mit den zusätzlichen Platzhaltern #copyRoot und #copySubs notieren lassen. Neben der Definition der offenen Ports erfolgt auch eine formale Darstellung der Bindung solcher offenen Ports, mit der sie in nicht-offene Ports überführt werden können. Dies führt uns schließlich zum Einsatz offener Ports bei Transitionstypen, ihrer Bindung bei der Verwendung der Typen für konkrete Transitionen und dem *Migrationssatz*, mit dem sich die konsistente Migration eines XML-Dokuments über eine solche typisierte Transition beweisen lässt.

Def. 6.11: Die Platzhalter bei offenen Ports

$$\begin{aligned} \text{CopyPlaceholder} := \{\ & (\#\text{copyRoot}, \text{rootExceptions}), \\ & (\#\text{copySubs}, \text{rootExceptions}, \text{subExceptions}) \mid \\ & \text{rootExceptions}, \text{subExceptions} \subseteq \text{XmlElements}\ \} \end{aligned}$$

Wie bereits in Abschnitt 6.3.1 erläutert, können bei offenen Ports die Platzhalter #copyRoot und #copySubs verwendet werden. Dabei kann entweder nur #copyRoot allein oder #copyRoot zusammen mit #copySubs stehen. Der erste Fall wird durch das erste Element der Menge CopyPlaceholder, der zweite Fall durch das zweite Element der Menge repräsentiert. Das Schlüsselwort #copyRoot kann in den Dokumenttypen eines offenen Ports statt eines Wurzelelements angegeben werden. Ein solcher Dokumenttyp legt dann kein konkretes Wurzelelement fest, sondern bestimmt höchstens eine Menge obligatorischer Unterelemente. Später kann der Platzhalter an eine Menge von konkreten XML-Elementen gebunden werden, indem er durch die Wurzelelemente eines anderen Ports ersetzt wird (vgl. Def. 6.14). Zusätzlich kann mit dem Platzhalter #copySubs auch die Übernahme von Unterelementen gefordert werden. Ergänzt werden die Schlüsselwörter jeweils um eine Teilmenge von XmlElements, mit der, wie bereits bei der Erläuterung offener Ports in Abschnitt 6.3.1 dargelegt, die Ausnahmen beim Kopieren der Elemente spezifiziert werden können.

Syntax: Das Element #copyRoot aus CopyPlaceholder wird durch Ableitung des Nicht-Terminals ‚CopyRoot' der EBNF-Grammatik notiert, das Element #copySubs analog durch das Nicht-Terminal ‚CopySubs'. Die Ableitungen führen jeweils zu den terminalen Symbolen ‚#copyRoot' bzw. ‚#copyRoot_except', falls Ausnahmen aufgelistet werden sollen (rootExceptions $\neq \varnothing$), analog zu ‚#copySubs' und ‚#copySubs_except'.

Def. 6.12: Offene Dokumenttypen

$$\begin{aligned}\text{OpenDocTypes} := \{\ &(\text{root},\text{subs}) \mid \\ &\text{root} \in \text{XmlElements} \cup \text{AnyPlaceholder} \cup \text{CopyPlaceholder}, \\ &\text{subs} \subseteq \text{XmlElements}\ \} \\ = (\text{XmlElements} &\cup \text{AnyPlaceholder} \cup \text{CopyPlaceholder}) \times \mathcal{P}(\text{XmlElements})\end{aligned}$$

Bei einem offenen Dokumenttyp kann durch den Gebrauch von Platzhaltern, die später an konkrete Werte gebunden werden, die Dokumentmenge der konformen XML-Dokumente zunächst offen gehalten werden. Als Platzhalter sind dafür zusätzlich #copyRoot und #copySubs aus der Menge CopyPlaceholder (vgl. Def. 6.12) erlaubt. Offene Dokumenttypen werden bei der Definition offener Ports benötigt (siehe Def. 6.13).

Da die Menge der offenen Dokumenttypen eine echte Erweiterung der Menge der gewöhnlichen Dokumenttypen ist, gilt die Beziehung:

$$\text{DocTypes} \subset \text{OpenDocTypes}$$

Syntax: Ein Element aus OpenDocTypes wird durch Ableitung des Nicht-Terminals ‚OpenDoctype' der EBNF-Grammatik notiert. Sei dazu ein beliebiger, offener Dokumenttyp type gegeben, mit type = (root, subs) ∈ OpenDocTypes \ DocTypes und subs = $\{s_1, ...s_n\}$. Dann schreibt man:

a) "#copyRoot_except(re_1, ...re_m) [s_1, ...s_n]"

falls root = (#copyRoot, rootExceptions) ∈ CopyPlaceholder
und rootExceptions = {re_1, ...re_m}

b) "#copyRoot_except(re_1, ...re_m) [#copySubs_except(se_1, ...se_l), s_1, ...s_n]"

falls root = (#copySubs, rootExceptions, subExceptions) ∈ CopyPlaceholder
und rootExceptions = {re_1, ...re_m} und subExceptions = {se_1, ...se_l}

c) Das Postfix "_except(…)" kann entfallen, wenn die Menge der Ausnahmen leer ist, d.h. rootExceptions = Ø bzw. subExceptions = Ø.

Beispiele: #copyRoot [http://namespace1 elementA, http://namespace2 elementB]
#copyRoot [#copySubs_except(elementA), elementB]
#copyRoot_except(elementC) [ ]

Def. 6.13: Offene Ports

$$\text{OpenPorts} := \mathcal{P}(\text{OpenDocTypes})$$

Ein offener Port op∈OpenPort ist ein Port, bei dem durch den Gebrauch von Platzhaltern, die später an konkrete Werte gebunden werden, die Dokumentmenge der konformen XML-Dokumente offen gehalten werden kann. Die Platzhalter wurden

durch die hier verwendeten offenen Dokumenttypen eingeführt. Aus der Beziehung DocTypes ⊂ OpenDocTypes folgt direkt die Beziehung:

$$\text{Ports} \subset \text{OpenPorts}$$

Syntax: Wie im Nicht-Terminal ‚OpenPort' der EBNF-Grammatik aus Abschnitt 6.3.1 definiert, wird ein offener Port schlicht durch die Auflistung der enthaltenen Dokumenttypen notiert.

Um die Bindung eines offenen Ports an einen nicht-offenen Port zu definieren, wollen wir zunächst die Bindung eines offenen Dokumenttyps mit Platzhaltern an einen nicht-offenen Dokumenttyp als Hilfsfunktion definieren. Dazu müssen mehrere Fälle unterschieden werden:

i) Wenn der offene Dokumenttyp nur den Platzhalter #copyRoot trägt und das zu bindende Wurzelelement nicht in die Menge der Ausnahmen fällt, übernehme dieses Wurzelelement anstelle des Platzhalters.
ii) Wenn der offene Dokumenttyp beide Platzhalter #copyRoot und #copySubs enthält und das zu bindende Wurzelelement nicht in die Menge der Ausnahmen fällt, verfahre wie bei i) und übernehme zusätzlich die Unterelementliste abzüglich der spezifizierten Ausnahmen.
iii) Wenn das zu bindende Wurzelelement in die Menge der Ausnahmen fällt, wird das leere Wort (ε) zurückgegeben.

Als formale Definition ergibt sich somit:

Def. 6.14: Die Hilfsfunktion binding'

$$\text{binding}' : (\text{OpenDocTypes} \setminus \text{DocTypes}) \times \text{DocTypes} \longrightarrow \text{DocTypes}$$
$$\forall o = (\text{root}_o, \text{subs}_o) \in \text{OpenDocTypes}, \forall d = (\text{root}_d, \text{subs}_d) \in \text{DocTypes}:$$

$$\text{binding}'(o,d) = \begin{cases} (\text{root}_d, \text{subs}_o) & \text{falls } \text{root}_o \in \text{CopyPlaceholder} \\ & \text{mit } \text{root}_o = (\#\text{copyRoot}, \text{rootExceptions}) \\ & \wedge\, \text{root}_d \notin \text{rootExceptions} \\ (\text{root}_d, \text{subs}_o \cup (\text{subs}_d \setminus \text{subExceptions})) & \\ & \text{falls } \text{root}_o \in \text{CopyPlaceholder mit} \\ & \text{root}_o = (\#\text{copySubs}, \text{rootExceptions}, \text{subExceptions}) \\ & \wedge\, \text{root}_d \notin \text{rootExceptions} \\ \varepsilon & \text{falls } \text{root}_o \in \text{CopyPlaceholder mit} \\ & \wedge\, \text{root}_d \in \text{rootExceptions} \end{cases}$$

Für die Bindung eines offenen Ports an einen nicht-offenen Port wird die folgende Funktion binding definiert, die Dokumenttypen aus dem zu bindenden Port übernimmt, wenn sie keinen Platzhalter enthalten, und anderenfalls durch Aufruf der Hilfsfunktion binding' an alle Dokumenttypen bindet, die in dem nicht-offenen Port enthalten sind.

Als Ergebnis wird ein neuer Port zurückgeliefert, bei dem alle #copyRoot- und #copySubs-Platzhalter durch konkrete XML-Elemente ersetzt wurden.

Def. 6.15: Die Bindungsfunktion binding

$$\text{binding: OpenPorts} \times \text{Ports} \rightarrow \text{Ports}$$

$$\forall o \in \text{OpenPorts}\ \forall p \in \text{Ports:}$$

$$\text{binding}(o,p) := \{o_i \mid o_i \in (o \cap \text{DocTypes})\} \cup \{\text{binding}'(o_j,p_k) \mid o_j \in o \cap (\text{OpenDocTypes} \setminus \text{DocTypes}) \wedge (p_k \in p)\}$$

Transitionstypen und typisierte Transitionen

Wie in Abschnitt 6.3.2 dargelegt werden im statischen Modell Transitionstypen definiert, die zur Typisierung von Transitionen benutzt werden müssen. Demnach hat jeder Transitionstyp einen Quellport, einen Zielport und eine Transformationsvorschrift trans, die Dokumente aus der Dokumentenmenge des Quellports auf Dokumente aus der Dokumentenmenge des Zielports abbildet. Da es sich bei dem Zielport um einen offenen Port handelt, ist der Wertebereich der Funktion trans die Vereinigungsmenge aller sinnvollen Bindungen des Zielports. Sinnvoll sind in diesem Zusammenhang nur solche Ports, die zum Quellport kompatibel sind und damit überhaupt als Vorgänger im Kontrollfluss des Diagramms in Frage kommen. Somit ergibt sich folgende Definition für Transitionstypen:

Def. 6.16: Transitionstypen

$$\text{TTypes} := \{ (\text{source}, \text{target}, \text{trans}) \mid \text{source} \in \text{Ports}, \text{target} \in \text{OpenPorts}, \text{trans Transformationsfunktion kodiert in XSL.} \}$$

$$\text{trans} : L(\text{source}) \rightarrow \bigcup_{\substack{p \in \text{Ports} \\ p \subseteq \text{source}}} L(\text{binding}(\text{target},p))$$

$$(**)\ \forall x \in L(\text{source}),\ \forall p \in \text{Ports mit } p \subseteq \text{source}: \quad x \in L(p) \Rightarrow \text{trans}(x) \in L(\text{binding}(\text{target},p))$$

Die obige Randbedingung (**) an die Funktion trans besagt, dass alle XML-Dokumente aus der Dokumentmenge eines Ports p, der zum Quellport source des Transitionstyps kompatibel ist ($p \subseteq$ source), durch die Funktion trans so abgebildet werden, dass sie zur Dokumentmenge des Zielports target, gebunden an eben jenen Ausgangsport p, gehören. Diese Bedingung ist nötig, damit target, gebunden an p, Ausgangsport einer Transition ist, wenn p der Eingangsport ist, und die Transformationsfunktion dann auch tatsächlich zwischen diesen Ein- und Ausgangsports abbildet. Die folgende Definition von typisierten Transitionen stellt einen solchen Fall dar, wobei der Eingangsport input die Rolle des Ports p übernimmt:

Def. 6.17: Typisierte Transitionen

$$\begin{aligned}\text{Transitions} := \{\ (\text{input}, \text{output}, \text{type}) \mid\ & \\ & \text{input} \in \text{Port}, \text{output} \in \text{Port}, \\ & \text{type} = (\text{source}, \text{target}, \text{trans}) \in \text{TTypes}, \\ & \text{input} \subseteq \text{source} \wedge \\ & \text{binding}\ (\text{target}, \text{input}) \subseteq \text{output}\ \}\end{aligned}$$

Eine typisierte Transition besteht somit neben den konkreten Ein- und Ausgangsports aus einem Typ, wobei die Ein- und Ausgangsports von Transition und Transitionstyp durch die angegebenen Teilmengenbeziehungen zueinander kompatibel sein müssen. Die Ein- und Ausgangsports der Transition sind im Prozessdiagramm de facto durch die Ein- und Ausgangsports der Zustände gegeben, die diese Transition miteinander verbindet. Das XML-Migrationsdokument kann so entlang dieser Transition von einem Zustand zu seinem Nachfolgerzustand wandern. Der folgende *Migrationssatz* beweist, dass mit Hilfe der im Transitionstyp festgelegten Transformationsfunktion trans jedes Dokument, das zur Dokumentmenge des Eingangsports gehört, auch konform zum Ausgangsport der Transition gemacht werden kann.

Satz 6.18: Migrations-Satz

$\forall$ t=(input, output, type)$\in$Transitions, type=(source, target, trans)$\in$TTypes:

$$\forall\ x \in L(\text{input}):\quad \text{trans}(x) \in L(\text{output})$$

Beweis:

Sei $x \in L(\text{input})$ beliebig. Z.z.: $\text{trans}(x) \in L(\text{output})$

	Es gilt: $\text{input} \subseteq \text{source}$	(6.17)
	Aus $x \in L(\text{input})$ folgt damit:	
$\Rightarrow$	$x \in L(\text{source})$	(6.10)
$\Rightarrow *$	$\text{trans}(x) \in L(\text{binding}(\text{target}, \text{input}))$	(6.16 $**$)
	Es gilt: $\text{binding}(\text{target}, \text{input}) \subseteq \text{output}$	(6.17)
$\Rightarrow$	$L(\text{binding}(\text{target}, \text{input})) \subseteq L(\text{output})$	(6.10)
$\Rightarrow$	$\text{trans}(x) \in L(\text{output})$	($*$)

Damit ist gezeigt, dass Transformationsfunktionen auf Transitionen nicht zu Konsistenzbrüchen führen können, wenn sie der angegebenen Randbedingung (6.16 $**$) genügen.

Die hiermit abgeschlossenen Betrachtungen zur formalen Definition von Ports haben für alle syntaktischen Elemente zur Notation von Ports (vgl. Abschnitt 6.3.1) eine passende Semantik definiert und diese in Bezug zum Konzept der Dokumentmigration (vgl. Abschnitt 6.1.1) und der typisierten Transitionen (vgl. Abschnitt 6.3.2) gesetzt. Dadurch ist ein formales Gesamtkonzept für die Modellierung der Dokumenttypen

durch Ports und offene Ports, die Bindung offener Ports sowie den Einsatz typisierter Transitionen entstanden, das als Grundlage für die Implementierung des Typkonzepts im Metamodell (vgl. Abschnitt 7.4), im Modellierungswerkzeug (vgl. Kapitel 9) und im Prozessinterpreter (vgl. Kapitel 10) benutzt wird.

6.4 Bewertung

Nachdem in den vorherigen Abschnitten das Konzept der XML-Prozesse vorgestellt und ihre Modellierung mit Hilfe von Prozessdiagrammen beschrieben wurde, soll an dieser Stelle eine Evaluierung stattfinden. Dabei soll überprüft werden, inwieweit durch diesen Ansatz die im Abschnitt 3.4 vorgenommene Zielsetzung erfüllt wird. Im Hinblick auf den Einsatzbereich E-Business waren die beiden wichtigsten Anforderungen an ein Workflow-Management-System die Integration heterogener Back-End-Systeme und die XML-Unterstützung bezüglich der Workflow-relevanten Daten.

Anforderung	**Prozessdiagramme**
Visuelle Sprache	+
Allgemein akzeptiert	+
Leicht/Intuitiv verständlich	+
Ablauforientiert	+
Alle Basiskontrollfluss-Konstrukte	+
Ein Start- und ein Endpunkt	+
XML-Dokumentfluss	+
Zugriff auf XML-Dokumente	+
Integration von Softwarekomponenten	+
Parametrisierung der Komponenten	+
Geschachtelte Prozesse (Hierarchie)	+
Statisches Modell der Komponenten	+
Transaktionale Prozesse	+
Präzise/Eindeutige Semantik	+

Tab. 6.3: Bewertung von Prozessdiagrammen

Die Forderung nach der Einbindung vorhandener Softwarekomponenten wurde für XML-Prozesse durch das im Abschnitt 6.1.2 eingeführte Konzept der Komponentenintegration mittels Konnektoren umgesetzt. Dadurch ist das WFMS in der Lage, nahezu beliebige Systeme anzusprechen. Das Problem der heterogenen Schnittstellen wird für das WFMS transparent innerhalb der Konnektoren gelöst. Um solche Komponenten in einem Workflow-Modell zu verwenden, bietet die Modellierungssprache geeignete Konstrukte. Dabei ist es nicht nur möglich, bereits vorhandene Systeme als Aktivitäten zu verwenden. Zunächst kann ein abstraktes Modell des Workflows entworfen werden, um später den einzelnen Aktivitäten konkrete Ausführungsinstanzen zuzuordnen. Auch die Übergabe von Parametern an die eingesetzten Komponenten wird im Ansatz der XML-Prozesse berücksichtigt.

Der geforderten Verarbeitung von XML-Dokumenten wurde durch die Anlehnung an das Dokumentmigrationsmodell (vgl. Abschnitt 6.1.1) Rechnung getragen. Dabei migriert ein XML-Dokument, das die Workflow-relevanten Daten beinhaltet, durch die Aktivitäten des Prozesses. Da dieser Datenfluss implizit stattfindet, braucht er in den Prozessdiagrammen nicht explizit modelliert zu werden. Der Zugriff auf die XML-Daten geschieht mit Hilfe von Anfragesprachen wie XPath beispielsweise innerhalb von Guards. Dabei wird durch die im Abschnitt 6.3 beschriebene Modellierung der Dokumentstrukturen die Konsistenz der Prozessmodelle bezüglich der XML-Daten sichergestellt.

Da Prozessdiagramme auf UML-Aktivitätendiagrammen basieren, bieten sie alle Vorteile von UML. Dazu zählt vor allem die Mächtigkeit der Kontrollflusskonstukte wie Parallelität. Als zusätzliche Möglichkeit neben den bereits vorhandenen, wurde ein Sprachelement eingeführt, um Prozesse als Transaktionen ablaufen zu lassen, da dies in der Praxis häufig sichergestellt werden muss. Abschließend wurde gefordert, dass jedes Workflow-Modell eine eindeutige Semantik besitzt. Dies wurde in Prozessdiagrammen durch bestimmte Einschränkungen der Aktivitätszustände erreicht, womit die Grundvoraussetzung für ihre vollständig maschinelle Ausführung gegeben ist. Bevor wir Überlegungen in dieser Richtung anstellen, soll im nächsten Kapitel zunächst die Modellierungssprache für Prozessdiagramme formal spezifiziert werden.

7 Metamodell für Prozessdiagramme

Die im vorherigen Kapitel vorgestellten Erweiterungen von UML-Aktivitätendiagrammen sollen formal als Erweiterungen des UML-Metamodells definiert werden. Dazu erläutert der folgende Abschnitt zunächst die erforderlichen Grundlagen des allgemeinen UML-Metamodells. Danach folgt ein Einblick in die Erweiterungsmechanismen, die die UML-Spezifikation zur Ergänzung des Metamodells zur Verfügung stellt. Unter Ausnutzung dieser Erweiterungsmechanismen beschreibt Abschnitt 7.3 das speziell zur Repräsentation von Prozessdiagrammen erweiterte Metamodell als eigenes UML-Profil für Prozessdiagramme. Der vierte Abschnitt dokumentiert die Implementierung dieses Metamodells in der Programmiersprache Java, die als Grundlage für die im dritten Teil der Arbeit vorgestellten Werkzeuge fungiert. Daran anschließend zeigen die Abschnitte 7.5 und 7.6 Algorithmen für dieses Metamodell, mit denen sich Prozesse validieren und Ports automatisch berechnen lassen.

7.1 Das UML-Metamodell

Bei dem sogenannten *UML-Metamodell* handelt es sich laut [Omg01] (2-3) um ein implementierungsunabhängiges Modell, das zur deklarativen Festlegung der Semantik aller UML-Sprachelemente dient. Jedes Element wird dabei unter drei Sichtweisen betrachtet. Durch einen *abstrakten Syntaxgraphen*, welcher selbst aus einem UML-Klassendiagramm besteht, erfolgt die Festlegung der Syntax aller UML-Elemente in ihrem jeweiligen Kontext. Auf diese Weise definiert sich die UML nach dem Prinzip des „Bootstrapping“ durch Klassendiagramme, die ihrerseits UML-Bestandteil sind. Die *Semantik* wird durch eine textuelle Beschreibung zu jedem Sprachelement erläutert. Schließlich werden mit Hilfe der Object Constraint Language (OCL, siehe Abschnitt 7.1.3) *Wohlgeformtheitsregeln* aufgestellt, die bei der Benutzung der Elemente einzuhalten sind, um beispielsweise bestimmte problematische Konstellationen von Sprachelementen zu verbieten.

7.1.1 Architektur von UML

Das UML-Metamodell stellt eine von vier Schichten einer Metamodell-Schichtenarchitektur dar:

- Das *Meta-Metamodell* legt die grundlegende Infrastruktur fest, um überhaupt die Darstellung von Sachverhalten mit grafischen Elementen zu ermöglichen. Dazu zählen die Definitionen für „Klasse“, „Assoziation“ und „Operation“ ([Omg01] 2-5).
- Mit Hilfe der Basiskonstrukte des Meta-Metamodells erfolgt nun die Beschreibung des *Metamodells*, also der gesamten UML-Sprache.
- Ein *Modell* ist eine konkrete Instanz des Metamodells, mit der beispielsweise ein reales Softwaresystem durch verschiedene Diagrammtypen spezifiziert wird.
- Beim *Benutzermodell* schließlich handelt es sich um die Ausprägung des Modells, also den Objektgraphen des Softwaresystems zu einem bestimmten Zeitpunkt der Ausführung.

Da es sich beim UML-Metamodell um eine komplexe Beschreibung vieler verschiedener Sprachelemente handelt, ist es in eine Reihe von *Packages* gegliedert, die jeweils eine logische Einheit bilden, wie beispielsweise zur Definition aller Elemente eines bestimmten Diagrammtyps. Im folgenden Abschnitt sollen die wichtigsten Bestandteile des Metamodells vorgestellt werden, die relevant für Aktivitätendiagramme sind und für die Erstellung von Prozessdiagrammen erweitert werden müssen.

7.1.2 Metamodell für Aktivitätendiagramme

Aktivitätendiagramme werden in der UML-Spezifikation als Spezialfall von Zustandsmaschinen angesehen. Aus diesem Grund beruht auch ihre Definition im Metamodell auf der von Zustandsmaschinen, welche um zusätzliche Elemente ergänzt wird. Abbildung 7.1 zeigt einen Ausschnitt aus dem abstrakten Syntaxgraphen des Metamodells mit den wichtigsten Meta-Elementen von Aktivitätendiagrammen.

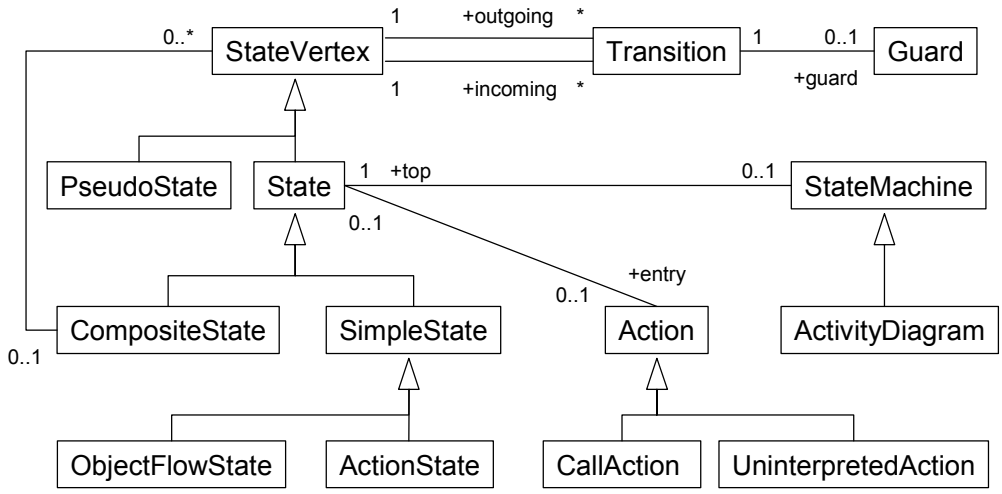


Abb. 7.1: Metamodell für Aktivitätendiagramme

Danach besitzt jede Zustandsmaschine (StateMachine) und damit auch jedes Aktivitätendiagramm genau einen Zustand (State) in der obersten Ebene (vgl. [Omg01] 2-152). Bei Zuständen werden zwei Arten unterschieden, CompositeState und SimpleState. Ein *CompositeState* besitzt selbst wieder beliebig viele Unterzustände und erlaubt damit die Schachtelung von Diagrammen, d.h. das Einbetten von Unterdiagrammen in einen Aktivitätszustand. Als *SimpleState* wird in Aktivitätendiagrammen stets das Element *ActionState* verwendet oder *ObjectFlowState* zur Beschreibung des Objektflusses zwischen den Aktionen.

Die Spezifikation der Aktivität, die in einem Zustand ausgeführt werden soll, geschieht über die Assoziation vom Zustand zum Element *Action*. Diese Aktion wird ausgeführt, sobald der zugehörige Zustand betreten wird. In Aktivitätendiagrammen wird als Aktion meistens das Element *CallAction* verwendet, durch das die Methode eines Objekts aufgerufen wird. Alternativ kann auch *UninterpretedAction* verwendet werden, wenn die Aktivität durch einen informalen Text beschrieben werden soll. Dies erlaubt zwar das Modellieren von Abläufen auf hohem Niveau, verhindert jedoch die Möglichkeit einer maschinellen Interpretation des Diagramms.

Neben gewöhnlichen Zuständen gibt es in Aktivitätendiagrammen auch sogenannte *PseudoStates*. Dazu zählen Verzweigungen, Fork- und Joinzustände und der Startzustand, also Zustände, in denen keine explizite Aktion ausgeführt wird.

Jeder Zustandsknoten (StateVertex) eines Diagramms besitzt beliebig viele Eingangs- bzw. Ausgangstransitionen zu anderen Zuständen in Form des Elements *Transition*. Dabei können bestimmte Transitionen einen *Guard* besitzen, um eine Bedingung für die Ausführung des betroffenen Strangs angeben zu können. Auf diese Weise entsteht ein gerichteter Graph, der den gesamten Kontrollfluss zwischen den Zuständen beschreibt. Es ist jedoch zu beachten, dass nicht jeder Zustand beliebig viele Transitionen besitzen darf, obwohl dies im abstrakten Syntaxgraph erlaubt ist. Beispielsweise untersagt die UML-Spezifikation, dass ein Fork-Zustand mehr als eine Eingangstransition hat (siehe [Omg01] 2-162). Um solche zusätzlichen Regeln für den Aufbau von Diagrammen zu spezifizieren, dient die Object Constraint Language. Da sie in dieser Arbeit für die notwendigen Erweiterungen im Bezug auf Prozessdiagramme verwendet werden soll, werden ihre Grundkonzepte im folgenden Abschnitt erläutert.

7.1.3 Object Constraint Language

An vielen Stellen der UML-Spezifikation müssen einschränkende Aussagen über den Aufbau von Diagrammen gemacht werden, die sich nicht ohne Weiteres im abstrakten Syntaxgraphen beschreiben lassen. Damit für diese Wohlgeformtheitsregeln nicht auf eine informale Spezifikation ohne eindeutige Semantik zurückgegriffen werden muss, wurde von der OMG die *Object Constraint Language* (OCL) eingeführt. Sie kann nicht nur dazu verwendet werden, um Aussagen innerhalb des UML-Metamodells zu formulieren, sondern auch in der Modell- oder Benutzerschicht. Daher werden beispielsweise Guard-Bedingungen häufig als OCL-Ausdrücke angegeben. Ein wichtiges Ziel bei der Entwicklung der OCL war es, eine Sprache zu schaffen, die keine unanschauliche mathematische Syntax hat, sondern die vielmehr auch von Laien ohne fundiertes mathematisches Wissen leicht zu lesen und schreiben ist. Dabei ist zu beachten, dass OCL keine Programmiersprache, sondern eine reine Ausdruckssprache ohne Seiteneffekte ist, d.h. die Auswertung eines OCL-Ausdrucks kann keine Manipulationen am zu Grunde liegenden Modell auslösen.

Die Erläuterungen zur OCL sollen beispielhaft anhand des Klassendiagramms aus Abbildung 7.1 erfolgen. Dabei wird nur auf die OCL-Sprachkonstrukte eingegangen, die für diese Arbeit relevant sind. Vergleiche zu OCL auch [Omg01] (6-1ff). Eine formale Grammatik der Object Constraint Language in EBNF-Notation findet sich in [Omg01] (6-96ff).

Kontext: Zunächst muss in jedem OCL-Ausdruck bestimmt werden, auf welches Modellelement sich seine Aussagen beziehen. Dazu wird der Klassenname des Elements hinter dem Schlüsselwort `context` angegeben. Wenn sich eine Aussage also auf das Element PseudoState bezieht, sieht der Ausdruck folgendermaßen aus:

```
context PseudoState inv: ...
```

Das Wort `inv` gibt dabei an, dass es sich bei dem Ausdruck um eine Aussage über das angegebene Element handelt, die für alle Exemplare des Elementtyps erfüllt werden

muss. Innerhalb des Ausdrucks wird mit Hilfe des Schlüsselworts `self` auf die Instanz des Elements verwiesen.

Logische Operatoren: Zur Formulierung des Ausdrucks stehen die bekannten Verknüpfungsoperatoren durch folgende Schlüsselwörter zur Verfügung:

`or`	logisches ODER
`and`	logisches UND
`implies`	Folgerung/Implikation

Zugriff auf Attribute: Die Werte von Attributen eines Modellelements können durch einen Punkt, gefolgt vom Namen des Attributs abgefragt werden. Wenn sich beispielsweise eine OCL-Regel auf den Startzustand eines Aktivitätendiagramms beziehen soll, lässt sich dies durch folgenden Ausdruck realisieren:

```
context PseudoState inv:
(self.kind=#initial) implies ...
```

Dabei ist `#initial` eine vordefinierte Konstante für das Attribut `kind` des Elements PseudoState, die den Startzustand identifiziert.

Navigation über Assoziationen: Sehr häufig muss man innerhalb eines OCL-Ausdrucks über Assoziationen von Elementen navigieren. Dies geschieht analog zum Zugriff auf Attribute durch einen Punkt, gefolgt von der Rollenbezeichnung der gewünschten Assoziation. Bei Assoziationen, die nicht die Kardinalität 1 besitzen, ist der Rückgabewert vom vordefinierten OCL-Datentyp *Collection*. Für diesen existieren eine Reihe von Operationen wie *size* oder *select*, die im nächsten Absatz erläutert werden. Sie sind durch einen Pfeil-Operator aufrufbar. Mit ihrer Hilfe wird beispielsweise in der UML-Spezifikation festgelegt, dass der Startzustand eines Aktivitätendiagramms keine Eingangs- und nur genau eine Ausgangstransition besitzt:

```
context PseudoState inv:
(self.kind = #initial) implies
((self.incoming->isEmpty())
 and (self.outgoing->size() = 1))
```

OCL-Mengenoperationen: Der vordefinierte Datentyp *Collection* bietet eine Reihe von Operationen, um auf die Elemente der Menge oder Liste zuzugreifen. Die wichtigsten und später benutzten Mengenoperationen sollen in der nachstehenden Tabelle kurz vorgestellt werden (vgl. [Omg01] 6.8.2.1).

`size():Integer`	Liefert die Anzahl der Element in der Collection.
`isEmpty():Boolean`	Liefert true, wenn die Collection leer ist.
`exists(expr:OclExpression):Boolean`	Liefert true, wenn es in der Collection wenigstens ein Element gibt, für das der angegebene Ausdruck wahr wird.
`one(expr:OclExpression):Boolean`	Liefert true, wenn es in der Collection genau ein Element gibt, für das der angegebene Ausdruck wahr wird.
`select(expr:OclExpression):Set`	Liefert die Teilmenge der Elemente aus Collection, für die der angegebene Ausdruck wahr wird.
`forAll(expr:OclExpression):Boolean`	Liefert true, wenn der angegebene Ausdruck für alle Elemente der Colleciton wahr wird.
`includes(object:OclAny):Boolean`	Liefert true, wenn object in der Collection enthalten ist.
`includesAll(c2:Collection):Boolean`	Liefert true, wenn alle Elemente aus c2 in der Collection enthalten sind.

Tab. 7.1: OCL-Mengenoperationen

Definition von Variablen und Funktionen: Um OCL-Ausdrücke übersichtlicher zu gestalten bzw. bestimmte Teilausdrücke wiederverwenden zu können, lassen sich mit Hilfe des Schlüsselworts *let* Variablen und Funktionen definieren. Diese erhalten durch die Definition einen eindeutigen Namen, unter dem sie in Ausdrücken verwendet werden können. Um beispielsweise die Anzahl der Ausgangstransitionen eines Zustands als Variable zu definieren, dient folgende Anweisung:

```
Context State inv:
let outCount: Integer = self.outgoing->size()
```

OCL-Typen: Neben einigen Basistypen sind auch alle Klassifizierer aus einem UML-Modell wie Klassen oder Interfaces Typen, die in OCL-Ausdrücken benutzt werden können. Für den Umgang mit Typen bietet OCL einige sehr hilfreiche Funktionen an, die auf jeder Objektinstanz aufgerufen werden können (vgl. [Omg01] 6.8.1.2):

`object.oclIsKindOf(type:OclType)` `:Boolean`	Wahr, falls einer der Typen oder Supertypen (transitiv) von object gleich type ist.
`object.oclIsTypeOf(type:OclType)` `:Boolean`	Wahr falls type einer der Typen von object ist.
`object.oclAsType(type:OclType)` `:type`	Liefert object, aber als Ausprägung des Typs type. Das Ergebnis ist nicht definiert, falls object nicht von diesem Typ oder einem seiner Subtypen ist.

Tab. 7.2: OCL-Hilfsfunktionen für das Arbeiten mit Typen

Die Object Constraint Language besitzt eine Reihe weiterer Möglichkeiten wie beispielsweise das Formulieren von Vor- und Nachbedingungen von Methoden. Da diese jedoch in dieser Arbeit nicht zum Einsatz kommen, soll auf eine tiefergehende Darstellung verzichtet werden.

7.2 UML-Erweiterungsmechanismen

Die UML bietet mit ihren verschiedenen Diagrammarten und Sprachkonstrukten umfangreiche Möglichkeiten zur Modellierung von Systemen, seien es Hardwaresysteme, Softwaresysteme oder zum Beispiel ganze Unternehmen. Neben den durch das UML-Metamodell definierten Standardkonzepten sieht das Metamodell auch vor, dass sich Erweiterungen der Sprache integrieren lassen (siehe [Omg01] 2.6). Dadurch lässt sich die UML flexibel für besondere Bedürfnisse und Anwendungsfelder anpassen. Mit den in diesem Abschnitt vorgestellten Konstrukten *Stereotype, Tag Definition* mit *Tagged Values* und *Constraint* hat man Basiskonzepte zur Hand, um die UML durch neue Modellelemente oder neue Semantiken zu ergänzen. Wenn es sich dabei um additive, also ergänzende Änderungen der Sprache handelt, die nicht im Widerspruch zur Standardsemantik stehen, spricht man von einer *UML-Erweiterung (UML Extension*, siehe [Omg01] 2-74*).*

Im Gegensatz dazu ist es auch möglich, eine sogenannte *UML-Variante* zu definieren, die über die Verwendung der drei Erweiterungsmechanismen hinausgeht. Dafür müsste man jedoch explizit neue Meta-Elemente zum UML-Metamodell hinzufügen und die Sprachvariante somit durch ein eigenes Metamodell spezifizieren (siehe [Alh99] S.6, 19f). Diese Möglichkeit der Sprachmodifikation ist weniger restriktiv als der Gebrauch der vordefinierten Erweiterungsmechanismen und kann leicht zu Inkompatibilitäten mit der standardmäßigen UML-Definition führen. Daher wollen wir diesen Ansatz hier nicht weiter verfolgen, sondern im Folgenden die expliziten Erweiterungskonstrukte vorstellen, um mit ihnen später eine UML-Erweiterung für den Entwurf von Prozessdiagrammen zu entwickeln (vgl. Abschnitt 7.3).

Die drei Erweiterungsmechanismen Stereotype, Tagged Values und Constraint sind fest im Metamodell der UML verankert. Sie erlauben es, die vorhandenen Meta-Elemente der UML flexibel anzupassen und somit für bestimmte Prozesse oder Implementierungssprachen UML-Erweiterungen zu definieren. Neben dem Ergänzen der semantischen Informationen lassen sich so auch nicht-semantische, also etwa syntaktische Änderungen vornehmen. Die gesamte Menge der für ein bestimmtes Anwendungsgebiet definierten Spracherweiterungen werden schließlich zu einem *Profil* zusammengefasst (siehe [Omg01] 2-74) und dem in dem speziellen Anwendungsfeld tätigen Modellierer zur Verfügung gestellt. Der folgende Auszug aus dem UML-Metamodell verdeutlicht in einem ersten Überblick die Zusammenhänge zwischen den einzelnen Erweiterungsmechanismen, auf die in den folgenden Abschnitten noch näher eingegangen wird.

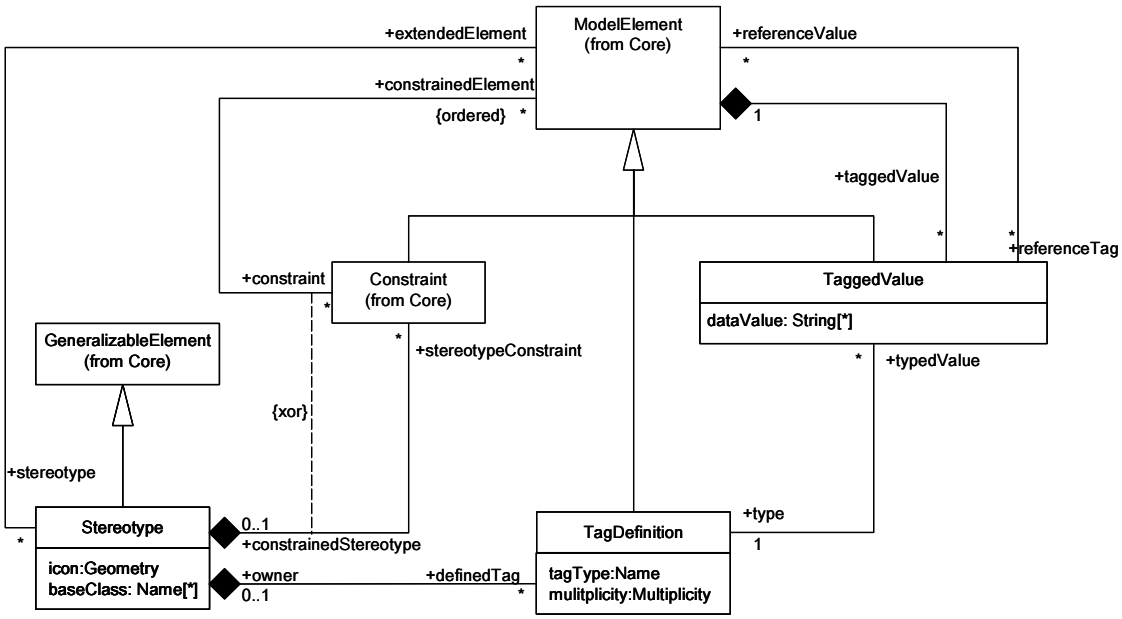


Abb. 7.2: Erweiterungsmechanismen der UML (aus [Omg01], 2.6.2)

7.2.1 Stereotypen

Das wohl wichtigste Konstrukt zur Spracherweiterung sind die Stereotypen ([Omg01] 2.6.3.2). Mit ihnen lassen sich neue, „virtuelle“ Metaklassen mit eigenen Attributen und Semantiken definieren und dem erweiterten Metamodell hinzufügen (vgl. [Alh99] S.12f). Auf der Modell-Ebene können die so definierten Stereotypen zur Klassifizierung der Elemente benutzt werden. Aus diesem Grund sind auch Namenskonflikte mit anderen Elementen der Klassifizierungshierarchie nicht erlaubt. Jedes Stereotyp basiert auf einem schon existierenden, im Metamodell wohldefinierten Element, der sogenannten *Basisklasse*. Von diesem Basiselement erbt das Stereotyp gleichsam alle Attribute, Operationen und Assoziationen. Allerdings kann es diese um weitere Attribute (Tag Definitions), semantische Bedingungen (Constraints) und optional um eine neue grafische Darstellung ergänzen. Jedes Element der Modellebene, das mit einem Stereotyp klassifiziert wird, erhält dann alle diese zusätzlichen Eigenschaften. Mit einem Stereotyp erweitert man in der Regel die vorhandenen Meta-Elemente, um ihnen eine neue Bedeutung im Kontext der Spracherweiterung zuzuweisen.

Im Abschnitt 7.3.2 zum Beispiel wird das Stereotyp «serviceState» benutzt, um in einem Prozessdiagramm einem Zustandselement die Semantik einer komponentenbasierten Aktivität zuzuschreiben. Da die Basisklasse dieses Stereotyps das Meta-Element CallState ist, erhält eine Instanz des Stereotyps «serviceState» die Semantik des CallState-Elements, zusätzlich aber auch die für das Stereotyp definierten Attribute und Nebenbedingungen.

Stereotypen kann man in einer Vererbungshierarchie strukturieren. Dabei darf ein Stereotyp A nur von einem Stereotyp B erben, wenn die Basisklasse von B der Basisklasse von A entspricht oder eine Unterklasse davon ist. Das eine Stereotyp erbt dann alle Attribute (Tagged Values) und Bedingungen (Constraints) des anderen.

7.2.2 Tag Definitions und Tagged Values

Um Stereotypen mit Attributen zu versehen, wird der Mechanismus der sogenannten *Tag Definitions* in Verbindung mit den *Tagged Values* angeboten ([Omg01] 2.6.2.4). Mit ihnen lassen sich neben den Stereotypen auch andere Elemente des Metamodells um *Pseudo-Attribute* ergänzen. Auf den Wert eines Attributs, der als Tagged Value gespeichert wird, kann man dabei über ein eindeutiges Schlüsselwort (Tag) zugreifen. Die Bezeichnung der Schlüsselworte eines Modellelements muss daher eindeutig sein. In der aktuellen Version 1.4 der UML wird empfohlen, die Definition neuer Attribute nur für Stereotypen zu benutzen und diesen zuzuweisen. Aus Kompatibilitätsgründen zur Vorgängerversion 1.3 ist es aber auch weiterhin nicht ausgeschlossen, die Schlüssel-Wert-Paare beliebigen Modellelementen zuzuordnen.

Im Gegensatz zu UML 1.3 ist es jetzt außerdem möglich, die zu einem Schlüsselwort korrespondierenden Werte genau zu typisieren. Während früher nur der einheitliche, generische Typ „String“ benutzt werden konnte (siehe [Omg00] 2-73f), stehen nun alle Standard-Datentypen und sogar Referenzen auf andere Klassen des Metamodells zur Verfügung. Der Typ der Werte und die Kardinalität werden in der *Tag Definition* festgelegt. Diese Tag-Definition besitzt dann in der Regel eine Assoziation zu einem bestimmten Stereotyp.

Tag Definitions und Tagged Values sind somit ein adäquates Mittel, um im Zusammenhang mit Stereotypen zusätzliche Eigenschaften und Attribute für Modellelemente zu definieren, die zum Beispiel Informationen für die Codegenerierung oder für die spezielle neue Semantik tragen.

4.2.3 Constraints

Mit Hilfe von *Constraints* lassen sich durch Regeln – formuliert in einer adäquaten Sprache – semantische und syntaktische Ergänzungen für Modellelemente spezifizieren ([Omg01] 2.6.2.1). Diese Regeln definieren Nebenbedingungen, die ein mit dieser UML-Erweiterung formuliertes Modell einhalten muss, um wohlgeformt zu sein. Die Einhaltung muss dabei in jedem stabilen Systemzustand gewährleistet sein, das ist immer dann, wenn gerade keine atomare Operation ausgeführt wird. Die entsprechende Überprüfung wird dabei an das jeweilige Modellierungswerkzeug delegiert. Die Constraints können in einer beliebigen Sprache sowohl formal – z.B durch OCL oder eine Programmiersprache – als auch natürlichsprachlich angegeben werden. Die jeweilige Interpretation der Regeln sollte der Sprache inhärent sein. Constraints können allen Elementen des UML-Metamodells zugewiesen werden, insbesondere natürlich den Stereotypen. Wurde für ein Stereotyp durch ein Constraint eine entsprechende Regel definiert, so muss sie für alle Instanzen dieses Stereotyps eingehalten werden.

7.2.3 Profile

Nachdem für ein bestimmtes Anwendungsgebiet eine UML-Erweiterung als Menge von Stereotypen, Tagged Values und Constraints definiert worden ist, kann man sie in einem speziellen UML-Package zusammenfassen. Ein solches Package erhält das in UML vordefinierte Stereotyp «*Profile*». Zusätzlich kann man in einem Profil die für die Erweiterung neu eingeführten Datentypen spezifizieren sowie eine Teilmenge der UML-Sprachdefinition angeben, die bei der Anwendung der Erweiterung für das

spezielle Anwendungsfeld relevant ist. Reichen die drei Erweiterungsmechanismen nicht vollständig aus, lassen sich für ein Profil auch natürlichsprachliche Semantikinformationen und Erklärungen ergänzen. Neben den Constraints für bestimmte Modellelemente können allgemeine Wohlgeformtheitsregeln die Spracherweiterung abrunden (vgl. [Omg99] S.25 „Working Definition of a UML Profile“ und [Omg01] 2-198f). Im nächsten Abschnitt dieses Kapitels findet sich als Beispiel für eine UML-Erweiterung das von uns aufgestellte Profil zur Modellierung von Prozessdiagrammen.

7.3 UML-Profil für Prozessdiagramme

In diesem Abschnitt soll nun mit Hilfe der UML-Erweiterungsmechanismen die Sprache zur Modellierung von XML-Prozessen formal spezifiziert werden. Wie in Abschnitt 7.2.4 beschrieben wurde, lässt sich eine Menge von Erweiterungen der UML für ein bestimmtes Anwendungsgebiet zu einem sogenannten Profil zusammenfassen. Daher sollen die in den folgenden Abschnitten spezifizierten Sprachkonstrukte ein „Profil zur Modellierung von Prozessdiagrammen“ bilden. Folgende Erweiterungen bzw. Einschränkungen von Aktivitätendiagrammen sind notwendig, um damit Prozessdiagramme erstellen zu können:

- Jedes Prozessdiagramm hat genau einen Start- und Endzustand.
- Es müssen Ports definiert werden können, mit denen sich die Schnittstellen der Zustände und Transitionen beschreiben lassen. Jeder Port repräsentiert eine spezielle Menge von XML-Dokumenten, die zu diesem Port konform sind.
- In Aktivitätszuständen ist als einzige Aktion die Integration einer bestimmten Softwarekomponente als Service erlaubt.
- Die Schnittstelle eines Service wird durch die Angabe von Ports definiert.
- Für den Aufruf eines Service müssen Parameter angegeben werden können.
- Jede Transition ist als Objektfluss des XML-Dokuments zwischen den Aktivitäten zu interpretieren. Transitionen sind typisiert, wobei Instanzen eines Typs nur bestimmte Ports miteinander verbinden können.
- In Guards müssen auch XPath-Ausdrücke erlaubt sein, um auf das XML-Dokument zugreifen zu können. Die Ausdrücke müssen zu den jeweiligen Dokumenttypen kompatibel sein.
- Es muss ein Sprachkonstrukt eingeführt werden, um (Teil-)Prozesse als Transaktionen ausführen zu können.
- Damit eine Ausführung des Prozessmodells durch den Prozessinterpreter möglich ist, müssen zusätzliche Wohlgeformtheitsregeln eingehalten werden.

Das Profil für die Modellierung von Prozessdiagrammen besteht im Wesentlichen aus einigen Datentypen zur Realisierung des Port-Konzeptes und mehreren Stereotypen, die die oben genannten Erweiterungen umsetzen, sowie den zugehörigen Tag-Definitionen und Constraints. Die Datentypen zur Repräsentation der Ports sind aus der formalen, mengentheoretischen Betrachtung aus Abschnitt 6.3.3 abgeleitet. Sie werden zuerst vorgestellt. Danach schließt sich die detaillierte Definition der benutzten Stereotypen durch Tabellen an, wie sie in [Omg01] (3.35) vorgeschlagen werden.

7.3.1 Umsetzung des Port-Konzepts in UML-Datentypen

Die bei der formalen Definition von Ports in Abschnitt 6.3.3 definierten Mengen werden im Folgenden als Datentypen im UML-Metamodell repräsentiert. Jede dort definierte Menge entspricht dabei einem Datentyp, jedes Element einem Wert dieses Datentyps. Die nachstehende Abbildung zeigt die Datentypen in der grafischen UML-Notation. Sie werden in den folgenden Stereotyp-Definitionen als Attributtypen benutzt.

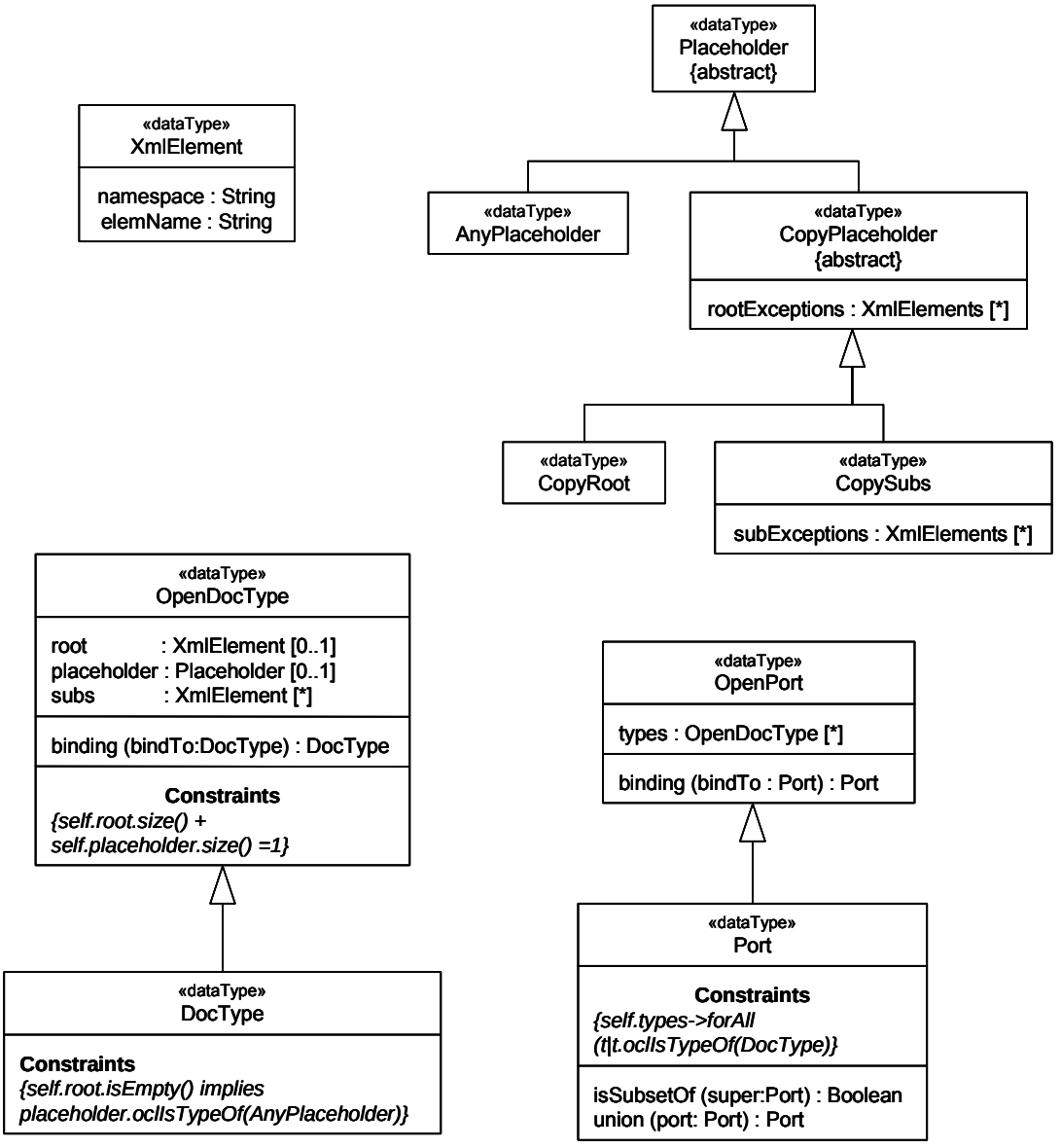


Abb. 7.3: UML-Datentypen zur Repräsentation von Ports

Die beiden Datentypen Placeholder und CopyPlaceholder sind als abstrakt gekennzeichnet, weil von ihnen keine konkreten Ausprägungen erzeugt werden können. Sie dienen lediglich der Generalisierung der anderen drei Platzhalter-Datentypen. Somit entspricht ein AnyPlaceholder-Exemplar dem #any-Platzhalter aus der in Abschnitt 6.3.3 definierten Menge ’AnyPlaceholder’. Die Elemente der Menge ’CopyPlaceholder’ werden durch die Spezialisierung auf die Datentypen CopyRoot und CopySubs noch einmal in die zwei möglichen Gruppen – nur mit dem Platzhalter

#copyRoot auf der einen und zusammen mit dem Platzhalter #copySubs auf der anderen Seite – aufgeteilt.

Insgesamt ergibt sich der folgende Zusammenhang zwischen den Ausprägungen der oben definierten Datentypen (linke Spalte) und den Mengendefinitionen aus Abschnitt 6.3.3 (rechte Spalte). Die OCL-Funktion allInstances() (siehe [Omg01] 6-30) liefert dabei jeweils alle zu einem Zeitpunkt verfügbaren Ausprägungen eines Typs. Mit der Teilmengen-Beziehung soll angedeutet werden, dass diese Ausprägungen quasi Elementen aus den jeweils angegebenen Mengen entsprechen:

XmlElement.allInstances()	⊆	XmlElements	(Def. 6.1)
AnyPlaceholder.allInstances()	⊆	AnyPlaceholder	(Def. 6.3)
DocType.allInstances()	⊆	DocTypes	(Def. 6.4)
Port.allInstances()	⊆	Ports	(Def. 6.7)
CopyPlaceholder.allInstances()	⊆	CopyPlaceholder	(Def. 6.11)
OpenDocType.allInstances()	⊆	OpenDocTypes	(Def. 6.12)
OpenPort.allInstances()	⊆	OpenPorts	(Def. 6.13)

Abb. 7.4: Zusammenhang zwischen Datentypen und Mengen

Die Vererbungsrelation von DocType zu OpenDocType führt dazu, dass jede DocType-Ausprägung gleichzeitig vom Typ OpenDocType ist. Dies spiegelt die in Def. 6.12 angegebene Relation „DocTypes ⊂ OpenDocTypes“ wider. Analoges gilt für die Vererbungsbeziehung zwischen Port und OpenPort (vgl. Def. 6.13).

Die Methode „binding (bindTo :DocType) :DocType“ des Datentyps OpenDocType ist eine Umsetzung der in Def. 6.14 definierten Hilfsfunktion binding’. Sie muss entsprechend dieser Funktion mathematisch korrekt implementiert werden. Analoges gilt für die Methode „binding (bindTo :Port) :Port“ des Datentyps OpenPort, die der Funktion aus Def. 6.15 entsprechen muss. Die Methode „isSubsetOf (super :Port) :Boolean“ des Datentyps Port liefert genau dann „true“ zurück, wenn die Teilmengen-Beziehung für Ports entsprechend Def. 6.9 erfüllt ist. Die Methode „union (port :Port) :Port“ dient zur Bildung eines Ports, der die Dokumenttypen zweier Ports vereinigt. Sie wird bei der Berechnung eines Ausgangsports von Oder-Zusammenführungen (siehe nächster Abschnitt) benötigt.

Für den Datentyp Port wird in OCL die folgende Hilfsfunktion is#any() definiert, die untersucht, ob ein Port allein aus der Angabe des Platzhalters #any besteht und damit L(p)=XmlDocs gilt. Ein solcher Port heißt *unbeschränkter Port*. Diese Funktion wird für spätere Wohlgeformtheitsregeln und Constraints benötigt.

```
context Port def:
    let is#any():Boolean =
        (self.types->size()=1 and
         self.types->one(root.oclIsTypeOf(AnyPlaceholder) and subs->isEmpty())
```

7.3.2 Definition von Stereotypen für Prozessdiagramme

Die für die Verwendung in Prozessdiagrammen vorgenommenen Erweiterungen an Syntax und Semantik der Standardelemente von UML-Aktivitätendiagrammen sollen durch die folgenden Definitionen von Stereotypen exakt definiert werden. Dazu zeigt Abbildung 7.5 alle neu eingeführten Stereotypen mit ihren jeweiligen Tags im Kontext ihrer Basisklassen aus dem UML-Metamodell. Im Anschluss daran folgt die detaillierte Definition der einzelnen Stereotypen und Attribute durch Tabellen.

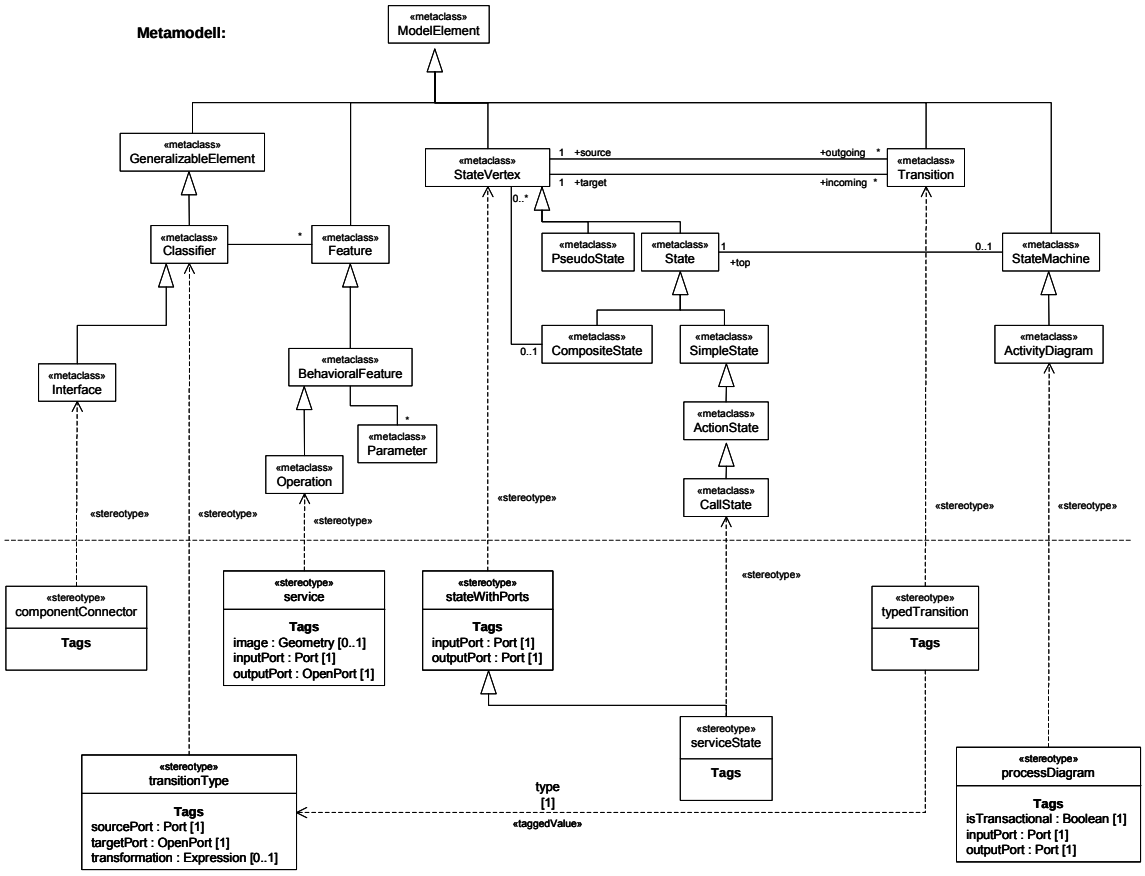
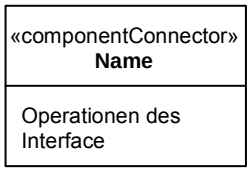


Abb. 7.5: Stereotypen des Profils

Elemente des statischen Modells

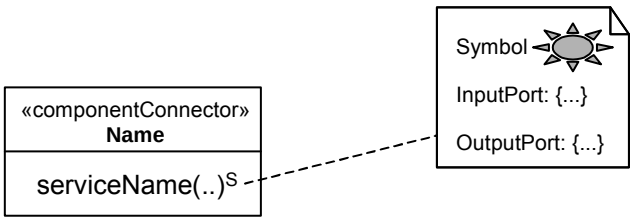
Das statische Modell spezifiziert die Konnektoren (vgl. Abschnitte 6.1.2 und 6.2) und die von ihnen angebotenen Services, mit denen bestimmte Softwarekomponenten in den Workflow integriert werden können. Ein Konnektor wird durch das Stereotyp «componentConnector» modelliert, das auf dem UML-Element Interface basiert. Dieses Element erlaubt die Definition von Operationen, die bei einer Realisierung des Interfaces von einer konkreten Konnektor-Klasse implementiert werden müssen.

Stereotyp	componentConnector
Basisklasse	UML::Core::Interface
Ober-Stereotyp	
Tag Definitionen	Keine
Einschränkungen (Constraints)	
Erläuterung	Interfaces, die als «componentConnector» klassifiziert sind, spezifizieren Konnektoren zu vorhandenen Softwarekomponenten. Eine Sammlung solcher Interfaces bildet das statische Modell für Prozessdiagramme (siehe «processDiagramm»). In den Konnektoren können bestimmte Operationen (siehe «service») definiert werden, die von Aktivitäten (siehe «serviceState») im XML-Prozess aufgerufen werden.
Darstellung	Darstellung des üblichen Symbols für Interfaces. Die sonst für Interfaces übliche Angabe «interface» wird durch «componentConnector» ersetzt. 

Tab. 7.3: Definition des Stereotyps «componentConnector»

Da die Operationen, die von einem «componentConnector» als Services zum Aufruf in Prozessdiagrammen angeboten werden (siehe Abschnitt 6.2), noch einige besondere Eigenschaften erfüllen müssen, werden sie als Stereotyp «service» definiert. Neben der üblichen Signatur der Operation benötigt eine Service-Definition die Angabe eines Eingabe- und eines Ausgabeports zur Spezifikation seiner Schnittstellen. Der Eingabeport gibt die Struktur der Eingabedokumente vor, die von dem Service verarbeitet werden können. Der Ausgabeport beschreibt die Menge der Dokumente, die nach Aufruf des Services an das nachfolgende Element weitergegeben werden (vgl. Abschnitt 6.3). Außerdem gibt es für jeden Service ein eigenes Symbol (siehe 6.2), das beim Einfügen eines Services in ein Prozessdiagramm benutzt werden kann.

Stereotyp	service
Basisklasse	UML::Core::Operation
Ober-Stereotyp	
Tag Definitionen	image inputPort outputPort
Einschränkungen (Constraints)	
Erläuterung	Ein Service definiert eine Operation, die in einer Aktivität eine Prozessdiagramms aufgerufen werden kann.

Darstellung	Übliche Darstellung einer Operation mit Signatur. Zur Unterscheidung von gewöhnlichen Operationen wird der Signatur ein hochgestelltes „S“ angefügt. Das dem Service zugewiesene Symbol und die Spezifikation der Ein- und Ausgabeports kann als UML-Kommentar an die Operation angefügt werden. Für gewöhnlich werden diese Attribute jedoch in spezifischer Weise von einem Modellierungswerkzeug erfasst und präsentiert. 

Tag	image
Stereotyp	service
Typ	UML::DataTypes::Geometry
Multiplizität	0..1
Erläuterung	Angabe eines Symbols, das für den Service im Falle eines Aufrufs in einem Prozessdiagramm benutzt werden kann.

Tag	inputPort
Stereotyp	service
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert die Menge von Dokumenten, die der zugehörige Service verarbeiten kann.

Tag	outputPort
Stereotyp	service
Typ	ProcessDiagrams::OpenPort
Multiplizität	1
Erläuterung	Dieser Port spezifiziert eine Menge von Dokumenten, die alle möglichen Ausgabedokumente dieses Service enthält.

Tab. 7.4: Definition des Stereotyps «service»

Prozessdiagramme

Da ein Prozessdiagramm ein spezielles Aktivitätendiagramm darstellt, wird hierfür ein eigenes Stereotyp definiert, dem jedes Diagramm entsprechen muss. Als Basisklasse dient das UML-Meta-Element ActivityGraph. Um die Forderung zu erfüllen, dass jedes Prozessdiagramm jeweils genau einen Start- und Endzustand hat (siehe Abschnitt 3.3, Punkt 2), sind dem Stereotyp entsprechende Constraints zugeordnet.

Bei der Festlegung der notwendigen Sprachkonstrukte wurde gefordert, dass Prozesse bzw. Teilprozesse als Transaktionen modelliert werden können (siehe Abschnitt 3.3, Punkt 8). Zu diesem Zweck besitzt jedes Diagramm ein Tag „isTransactional“, welches entweder den Wert TRUE annehmen kann, wenn Transaktionalität sichergestellt werden soll, oder FALSE, falls dies nicht notwendig ist. Um nur einen Teil eines Prozesses als Transaktion ausführen zu lassen, muss dieser als eigenes

Prozessdiagramm modelliert werden und mit Hilfe eines Subaktivitätszustands in das übergeordnete Diagramm eingebettet werden. Die folgende Tabelle zeigt die Definition des Stereotyps «processDiagram».

Stereotyp	processDiagram
Basisklasse	UML::ActivityGraphs::ActivityGraph
Ober-Stereotyp	
Tag Definitionen	isTransactional inputPort outputPort

Einschränkungen (Constraints)	1. Es gibt genau einen Initialzustand:
	`let topState :State = self.extendedElement.oclAsType(StateMachine).top`
	`let states :Set(State) = topState.oclAsType(CompositeState).subvertex`
	
	`states->one(oclIsTypeOf(PseudoState) and kind=#initial)`
	
	2. Es gibt genau einen Endzustand:
	`states->one(oclIsTypeOf(FinalState))`

Erläuterung	Aktivitätendiagramme, die als «processDiagram» klassifiziert sind, modellieren ein Prozessdiagramm, mit dem ein XML-basierter Workflow spezifiziert wird.
Darstellung	Falls der Eigenschaftswert von *isTransactional* wahr (TRUE) ist, wird der Startzustand mit dem Text „{transactional}“ versehen. Die Ports werden im Prozessdiagramm nicht dargestellt. Ein Modellierungswerkzeug hat dafür zu sorgen, dass sie passend repräsentiert werden und visualisiert werden können.

Tag	isTransactional
Stereotyp	processDiagram
Typ	UML::DataTypes::Boolean
Multiplizität	1
Erläuterung	Zeigt an, ob das zugehörige «processDiagram» einen Prozess modelliert, der Transaktionseigenschaften erfüllen muss.

Tag	inputPort
Stereotyp	processDiagram
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert die Menge von Dokumenten, die das zugehörige Prozessmodell verarbeiten kann.

Tag	outputPort
Stereotyp	processDiagram
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert eine Menge von Dokumenten, die alle möglichen Ausgabedokumente dieses Prozesses enthält.

Tab. 7.5: Definition des Stereotyps «processDiagram»

Dieses Stereotyp enthält außerdem die Tag-Definitionen für Eingangs- und Ausgangsport, so dass ein Prozessdiagramm auch als ein Element angesehen werden kann, das bestimmte Anforderungen an den Typ eines eingehenden Dokuments stellt

und nach der Ausführung bestimmte Dokumenttypen zurückliefert. Durch die Angabe der Ports lässt sich das Prozessdiagramm leicht als Prozessbaustein in einem anderen Workflow-Modell wiederverwenden (vgl. Abschnitt 6.3.1, Einbindung von Ports in Prozessdiagramme).

Zustände mit Schnittstellenspezifikation durch Ports

Alle Zustände in einem Prozessdiagramm werden von dem Migrationsdokument durchlaufen. Um ein möglichst hohes Maß an Konsistenz zu erreichen, werden zu jedem Zustand Ein- und Ausgabeschnittstellen in Form eines Ports angegeben (siehe Abschnitt 6.3). Durch den Zustand dürfen dann nur Dokumente fließen, die zu dem Eingangsport (inputPort) konform sind. Der Ausgangsport (outportPort) definiert die Struktur der Dokumente, nachdem sie durch diesen Zustand geflossen sind. Einige Zustände wie SynchStates oder ForkStates sind passiv und reichen ein eingehendes Dokument lediglich unverändert weiter, andere – vor allem die Aktivitäts- und Subaktivitätszustände – können das Dokument aktiv transformieren und nur ganz bestimmte Dokumenttypen verarbeiten.

Mit Hilfe der Ports an Zuständen ist es möglich, solche Dokumenttypen eindeutig zu modellieren. Allerdings gibt es für verschiedene Zustandsarten auch unterschiedliche Spielräume bei der Ausgestaltung der dort möglichen Ports. Daher findet bei den Constraints der folgenden Stereotyp-Definition, mit denen die Portverwendung eingeschränkt werden soll, eine Differenzierung nach dem jeweiligen Typ des Zustands statt. Die Bedingungen sollen dabei garantieren, dass die in der Tabelle 6.2 aus Abschnitt 6.3.1 festgehaltenen Vorgaben für die Ports bestimmter Zustandsarten eingehalten werden. Die Constraints ergeben sich unmittelbar aus den dort festgelegten Restriktionen.

Stereotyp	stateWithPorts
Basisklasse	UML::StateMachines::StateVertex
Ober-Stereotyp	
Tag Definitionen	inputPort ouputPort

Einschränkungen (Constraints)	1. Keine Einschränkungen, sondern lediglich Hilfsfunktionen zum Zugriff auf die Attribute:
	`context stateWithPorts def:`
	`  let inputPort:Port = self.definedTag`
	`  ->select(name='inputPort').typedValue.`
	`  referencedValue.oclAsType(Port)`
	`  let outputPort:Port = self.definedTag`
	`  ->select(name='outputPort').typedValue.`
	`  referencedValue.oclAsType(Port)`
	`  let state:StateVertex =`
	`  self.extendedElement.oclAsType(StateVertex)`
	`  let predecessors:Set(Port) = self.state.incoming.stereotype.`
	`  oclAsType(typedTransition).getBoundTarget()`
	
	2. Ein InitialState hat als Eingangsport den Eingangsport seines zugehörigen Prozessdiagramms:
	`if state.oclIsTypeOf(PseudoState) and`
	`  state.oclAsType(PseudoState).kind=#initial`
	`then inputPort = state.outgoing.stateMachine.`
	`  stereotype.oclAsType(processDiagram).`
	`  definedTag->select(name='inputPort').`
	`  typedValue.referenceValue.oclAsType(Port)`
	`endif`

3. Ein InitialState hat als Ausgangsport seinen Eingangsport:

```
if state.oclIsTypeOf(PseudoState) and
   state.oclAsType(PseudoState).kind=#initial
then outputPort = inputPort
endif
```

4. SynchStates, FinalStates, ForkStates, JoinStates und JunctionStates haben als Eingangsport den unbeschränkten Port:

```
if state.oclIsTypeOf(SynchState) or
   state.oclIsTypeOf(FinalState) or
   (state.oclIsTypeOf(PseudoState) and
    state.oclAsType(PseudoState).kind=#fork or
    state.oclAsType(PseudoState).kind=#join or
    state.oclAsType(PseudoState).kind=#junction)
then inputPort.is#any()
endif
```

5. SynchStates, FinalStates, ForkStates und DecisionStates haben als Ausgangsport den gebundenen Zielport ihrer eingehenden Transition:

```
if state.oclIsTypeOf(SynchState) or
   state.oclIsTypeOf(FinalState) or
   (state.oclIsTypeOf(PseudoState) and
    state.oclAsType(PseudoState).kind=#fork or
    (state.oclAsType(PseudoState).kind=#junction and
     state.incoming->size()=1))
then predecessors->forAll(p|p=outputPort)
endif
```

6. MergeStates haben als Ausgangsport die Vereinigungsmenge der gebundenen Zielports ihrer eingehenden Transitionen (siehe auch Abbildung 6.17):

```
if state.oclIsTypeOf(PseudoState) and
   state.oclAsType(PseudoState).kind=#junction and
   state.incoming->size()>1
then predecessors.types->forAll(t|outputPort.types->includes(t))
endif
```

7. JoinStates haben als Ausgangsport stets nur Dokumenttypen mit dem ausgezeichneten Wurzelelement „Join“. Jeweils ein Element aus den eingehenden gebundenen Zielports der eingehenden Transitionen bilden die Menge der obligatorischen Unterelemente von Join. Da jedes Element des einen Zielports mit jedem des anderen Zielports kombiniert werden muss, wird das kartesische Produkt gebildet (siehe auch Abbildung 6.18).

[Es ist bisher nicht gelungen, dieses Constraint in OCL auszudrücken. Eine Implementierung in Java liegt vor, kann aber nicht auf OCL übertragen werden, da OCL nicht die Bildung geschachtelter Mengen erlaubt (siehe [Omg01] 6.5.13). Die Bildung des kartesischen Produkts als OCL-Ausdruck ist hier schwierig, weil die Anzahl der in das Produkt einfließenden Mengen/Ports beliebig groß sein kann. Exemplarisch wird daher im folgenden Constraint davon ausgegangen, dass nur über genau zwei eingehende Transitionen vorgelagerte Ports vorhanden sind.]

```
predecessors->forAll(p1,p2| not p1=p2 implies
   p1.types->forAll(t1|
      p2.types->forAll(t2|
        outputPort.types->exists(doctype|
           doctype.root.elemName=”join” and
           doctype.subs->includes(t1.root) and
           doctype.subs->includes(t2.root) and
           doctype.subs->includesAll(t1.subs) and
           doctype.subs->includesAll(t2.subs)))))
```

8. Ein SubactivityState hat den gleichen Eingangsport wie der Startzustand des eingebetteten Prozesses:

```
if state.oclIsTypeOf(SubactivityState) then
   let embeddedInitial :PseudoState =
            state.oclAsType(SubactivityState).subvertex
            ->select(s|s.oclIsKindOf(PseudoState) and
                       s.oclAsType(PseudoState).kind=#initial)
   inputPort = embeddedInitial.stereotype.oclAsType(stateWithPorts).
               inputPort
endif
```

	9. Ein SubactivityState hat den gleichen Ausgangsport wie der Endzustand des eingebetteten Prozesses:
	`if state.oclIsTypeOf(SubactivityState) then`
	`  let embeddedFinal :FinalState =`
	`  state.oclAsType(SubactivityState).subvertex`
	`  ->select(s|s.oclIsKindOf(FinalState))`
	`  outputPort = embeddedFinal.stereotype.oclAsType(stateWithPorts).`
	`  outputPort`
	`endif`
	
	10. Ein FinalState hat als Ausgangsport den Ausgangsport des zugehörigen Prozessdiagramms:
	`if state.oclIsTypeOf(FinalState) then`
	`  outputPort = state.incoming.stateMachine.stereotype.`
	`  oclAsType(processDiagram).definedTag`
	`  ->select(name='outputPort').typedValue.`
	`  referencedValue.oclAsType(Port))`
	`endif`
	
	11. Ein ServiceState hat einen Eingangsport, der Teilport des Eingangsports ist, der für den angesprochenen Service im statischen Modell deklariert wurde (siehe Abbildung 6.16):
	`if state.oclIsTypeOf(CallState) then`
	`  inputPort.isSubsetOf (state.entry.oclAsType(CallAction).operation.`
	`  stereotype.oclAsType(service).definedTag`
	`  ->select(name='inputPort').typedValue.`
	`  referencedValue.oclAsType(Port))`
	`endif`
	
	12. Ein ServiceState hat einen Ausgangsport, für den der gebundene Ausgangsport, der für den angesprochenen Service im statischen Modell deklariert wurden, ein Teilport ist (siehe Abbildung 6.16):
	`if state.oclIsTypeOf(CallState) then`
	`  predecessors->forAll(pre|`
	`  (state.entry.oclAsType(CallAction).operation.`
	`  stereotype.oclAsType(service).definedTag`
	`  ->select(name='outputPort').typedValue.`
	`  referencedValue.oclAsType(OpenPort).`
	`  binding(pre)).isSubsetOf(outputPort))`
	`endif`

Erläuterung	Zustände, die als «stateWithPorts» klassifiziert sind, modellieren die Zustände in einem Prozessdiagramm. Die Ports dienen der Typisierung und Konsistenzprüfung im Zusammenhang mit dem XML-Fluss.
Darstellung	Die Ports werden im Prozessdiagramm nicht dargestellt. Ein Modellierungswerkzeug hat dafür zu sorgen, dass sie passend repräsentiert werden und visualisiert werden können.

Tag	inputPort
Stereotyp	stateWithPorts
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert die Menge von Dokumenten, die in diesen Zustand fließen können.

Tag	outputPort
Stereotyp	stateWithPorts
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert eine Menge von Dokumenten, die alle möglichen Ausgabedokumente dieses Zustandes enthält.

Tab. 7.6: Definition des Stereotyps «stateWithPorts»

Aktivitäten zur Komponentenintegration

Für die Aktivitäten im Prozessdiagramm wird das Stereotyp «serviceState» definiert. Es basiert auf der Basisklasse CallState, die dazu gedacht ist, die Methode einer Klasse aufzurufen (siehe Abschnitt 6.2.1). Bei einem «serviceState» wird die Einschränkung gemacht, dass nur eine als «service» deklarierte Operation aufgerufen werden darf (siehe Constraint 1). Beim Aufruf des Service müssen Werte für geforderte Parameter übergeben werden. Das UML-Metamodell (siehe [Omg01] 2.9.2.3) sieht als Wert für ein Argument einen beliebigen Ausdruck vor. Es wird keine Einschränkung bezüglich der möglichen Sprache getroffen. Somit sind für die Argumente auch Ausdrücke möglich, die für das WFMS verständlich sind und vor Aufruf der Methode ausgewertet werden können, um etwa Daten aus dem Migrationsdokument oder der Prozessumgebung (vgl. Abschnitt 6.2.2) zu übernehmen.

Stereotyp	serviceState
Basisklasse	UML::ActivityGraphs::CallState
Ober-Stereotyp	stateWithPorts
Tag Definitionen	keine

Einschränkungen (Constraints)	1. Die zugeordnete Operation muss vom Stereotyp «service» sein:
	`context serviceState inv:`
	`  let state : CallState = self.extendedElement.oclAsType(CallState)`
	`  state.entry.oclAsType(CallAction).operation.isStereotypedAs(service)`

Erläuterung	CallStates, die als «serviceState» klassifiziert sind, symbolisieren die Ausführung einer komponentenbasierten Aktivität. Jedem «serviceState» ist ein «service» zugeordnet, an den die Ausführung der Aktivität delegiert wird.
Darstellung	Standardmäßig wie ein CallState. Wenn der zugehörige «service» im Tag image ein Symbol spezifiziert, kann dieses zur Darstellung benutzt werden.

Tab. 7.7: Definition des Stereotyps «serviceState»

Die Ausführung der korrespondierenden Service-Operation in einem «serviceState» führt zu Änderungen auf dem Migrationsdokument. Deshalb müssen für einen «serviceState» bzw. die durch ihn definierte Service-Aktivität die erlaubten Eingangs- und die möglichen Ausgangsdokumenttypen durch die Angabe von Ports modelliert werden. Dies geschieht durch das Erben der entsprechenden Attribute von «stateWithPorts». Beim Setzen der Ports kann ein Werkzeug als Vorgabe die Werte für die Portattribute aus dem «service» übernehmen, der diesem «serviceState» zugeordnet ist (vgl. Abschnitt 9.2.5, Hinzufügen von Zuständen). Bei manchen Services kann es allerdings sein, dass ihre Portspezifikation die zulässigen Dokumentmengen nur sehr wage oder bei der Verwendung von Platzhaltern gar nicht einschränkt, weil bei diesen generischen Services die Dokumenttypen von konkreten Parameterwerten abhängen. In solchen Fällen können die Portspezifikationen für den «serviceState» vom Benutzer angepasst werden, nachdem die Werte der Parameter gesetzt sind (vgl. Abschnitt 6.3.1, Einbindung von Ports in Prozessdiagramme).

Typisierte Transitionen als XML-Dokumentfluss

Jedes Prozessdiagramm erhält bei seiner Ausführung ein XML-Dokument als Eingabe, das durch die einzelnen Aktivitätszustände migriert. Ein solcher Fluss von Daten zwischen den Aktivitäten lässt sich in Aktivitätendiagrammen durch Objektflüsse und Objektflusszustände beschreiben (siehe Abschnitt 6.2.3). In Prozessdiagrammen wird abkürzend auf die explizite Darstellung von Objektflüssen verzichtet und statt dessen ein spezielles Stereotyp «typedTransition» für Transitionen eingeführt, der ihnen neben der Kontrollfluss-Semantik auch die eines Dokumentflusses verleiht. Die Transitionen sind durch einen Typ «transitionType» klassifiziert, der durch einen Quellport, einen Zielport und eine Transformationsvorschrift spezifiziert wird (siehe Abschnitt 6.3.2). Der Quellport gibt an, welche Dokumenttypen über die Transition fließen können. Der Zielport hilft bei der Bestimmung der Zustände, deren Eingangsports zu dieser Transition kompatibel sind. Der Eingangsport eines nachfolgenden Zustands muss eine Obermenge des Zielports der Transition sein, damit alle von der Transition übergebenen Dokumente zum nachfolgenden Zustand konform sind. Mit der Transformationsvorschrift werden über den Quellport eingegangene Dokumente falls nötig so transformiert, dass sie konform zum Zielport des Transitionstyps werden (siehe 6.3.2 und 6.3.3: Migrationssatz). Um den Migrationssatz (Satz 6.18) anwenden zu können, muss die durch das Tag „transformation“ spezifizierte Transformationsfunktion die in Def. 6.16 getroffenen Einschränkungen erfüllen.

Stereotyp	transitionType
Basisklasse	UML::Core::Classifier
Ober-Stereotyp	
Tag Definitionen	sourcePort targetPort transformation

Einschränkungen (Constraints)	1. Keine Einschränkungen, sondern lediglich Hilfsfunktionen zum Zugriff auf die Attribute:
	`context transitionType def:`
	`let sourcePort:Port = self.definedTag->select(name='sourcePort').typedValue.referencedValue.oclAsType(Port)`
	`let targetPort:Port = self.definedTag->select(name='targetPort').typedValue.referencedValue.oclAsType(Port)`
	2. Einschränkungen an die Transformationsfunktion:
	Um die Anwendbarkeit des Migrationssatzes 6.18 sicherzustellen, muss die durch das Tag transformation spezifizierte Transformationsfunktion die in Def. 6.16 getroffenen Einschränkungen erfüllen.

Erläuterung	«transitionType» ist ein Stereotyp zur Definition von Transitionstypen. Die Attribute sourcePort und targetPort spezifizieren die Schnittstellen einer typisierten Transitionen, transformation die gegebenenfalls notwendige Konvertierungsfunktion zwischen diesen Ports.
Darstellung	Da die Basisklasse des Stereotyps Classifier keine eigenen Darstellung hat, wird ein «transitionType» in Anlehnung an eine Transition als Pfeil dargestellt, versehen mit dem Zusatz „«transitionType»“ und dem Namen des Classifiers. Der Pfeil verbindet nicht zwei Zustände, sondern wird kontextfrei dargestellt. Die Ports und die Transformationsfunktion werden im Prozessdiagramm nicht explizit gezeichnet. Möglich ist aber eine Darstellung als UML-Kommentar, oder ein Modellierungswerkzeug hat dafür zu sorgen, dass sie passend repräsentiert werden und visualisiert werden können.

Tag	sourcePort
Stereotyp	transitionType
Typ	ProcessDiagrams::Port
Multiplizität	1
Erläuterung	Dieser Port spezifiziert die Menge von Dokumenten, die in diese Transition fließen können.

Tag	targetPort
Stereotyp	transitionType
Typ	ProcessDiagrams::OpenPort
Multiplizität	1
Erläuterung	Dieser Port spezifiziert eine Menge von Dokumenten, die alle möglichen Ausgabedokumente dieser Transition enthält.

Tag	transformation
Stereotyp	transitionType
Typ	UML::DataTypes::Expression
Multiplizität	0..1
Erläuterung	Dieser Ausdruck spezifiziert eine Transformationsfunktion, die in der Regel als Stylesheet der Sprache XSLT kodiert ist. Der Definitionsbereich der Funktion entspricht dem Quellport des Transitionstyps, der Wertebereich dem Zielport des Transitionstyps. Der Ausdruck kann entweder das Stylesheet selbst enthalten oder auf eine externe Datei verweisen, in der das Stylesheet abgelegt ist. Die Transformationsfunktion entspricht der Funktion *trans* aus Def. 6.16 (siehe Abschnitt 6.3.3). Um die Anwendbarkeit des Migrationssatzes sicherzustellen, muss die Transformationsfunktion die in Def. 6.16 getroffenen Einschränkungen erfüllen.

Tab. 7.8: Definition des Stereotyps «transitionType»

Das folgende Stereotyp «typedTransition» definiert eine Transition, der ein solcher «transitionType» zugeordnet ist. Jede Transition eines Prozessdiagramms muss mit diesem Stereotyp klassifiziert sein. Die Einschränkungen des Stereotyps stellen die korrekte Umsetzung der Definition 6.17 einer typisierten Transition aus Abschnitt 6.3.3 sicher. Dort wird verlangt, dass der Eingangsport der Transition ein Teilport des Quellports des zugeordneten Transitionstyps ist (Constraint 2), und dass der gebundene Zielport des Typs ein Teilport des Ausgangsports der Transition sein muss (Constraint 3). Dies sind die Voraussetzungen zur Anwendung des Migrationssatzes (Satz 6.18), der einen konsistenten Dokumentfluss über die Transition sicherstellt.

Stereotyp	typedTransition
Basisklasse	UML::StateMachines::Transition
Ober-Stereotyp	
Tag Definitionen	type

Einschränkungen (Constraints)	1. Keine Einschränkungen, sondern lediglich Hilfsfunktionen zum Zugriff auf die Attribute:
	`context typedTransition def:`
	`  let type:transitionType =`
	`  self.definedTag->select(name='type').`
	`  typedValue.referencedValue.oclAsType(transitionType)`
	`  let inputPort:Port =`
	`  self.extendedElement.`
	`  oclAsType(Transition).source.stereotype`
	`  oclAsType(stateWithPorts).outputPort`
	`  let outputPort:Port =`
	`  self.extendedElement.`
	`  oclAsType(Transition).target.stereotype`
	`  oclAsType(stateWithPorts).inputPort`
	`  let boundTarget:Port =`
	`  self.type.targetPort.binding(self.inputPort)`
	
	`context typedTransition inv:`
	
	2. Der Ausgangsport des Quellknotens (=inputPort der Transition) ist ein Teilport des Eingangsports, der im Typ der Transition deklariert wurde:
	`  inputPort.isSubsetOf(type.sourcePort)`
	
	3. Der Ausgangsport des Transitionstypes, gebunden an den vorausgehenden Port (=boundTarget der Transition) ist ein Teilport des Eingangsports des Zielknotens der Transition (=outputPort der Transition):
	`  boundTarget.isSubsetOf(outputPort)`

Erläuterung	Transitionen, die als «typedTransition» klassifiziert sind, symbolisieren den Übergang von einem Zustand bzw. einer Aktivität im Prozessdiagramm zu einer anderen. Dabei charakterisieren sie sowohl den Kontroll- als auch den Datenfluss. Im Prozessdiagramm ist jedem Kontrollfluss implizit die Migration eines XML-Dokumentes als Datenfluss zugeordnet. Das XML-Dokument ist Ergebnis der Quellaktivität und Eingabe für die Zielaktivität. Eine «typedTransition» ist somit eine abkürzende Darstellung für einen *Object Flow State* mit dem XML-Dokument. Zusätzlich ist der Transition über ihren Typ eine Schnittstellenspezifikation und eine Konvertierungsvorschrift zwischen Quell- und Zielport zugeordnet.
Darstellung	Als Pfeil wie für gewöhnliche Transitionen mit der zusätzlichen Beschriftung «typedTransition» und dem Namen des zugewiesenen Typs. Die Erstellung eines Prozessdiagramms impliziert die Verwendung von «typedTransition» bei allen Transitionen. Die Beschriftung «typedTransition» kann daher zwecks besserer Überschaubarkeit entfallen.

Tag	type
Stereotyp	typedTransition
Typ	ProcessDiagrams::transitionType
Multiplizität	1
Erläuterung	Verweis auf den Typ einer Transition.

Tab. 7.9: Definition des Stereotyps «typedTransition»

Guard-Bedingungen

Um den Kontrollfluss eines Prozesses zur Laufzeit beeinflussen zu können, werden Ausdrücke einer XML-Anfragesprache wie XPath, XQL oder XQuery in Guards verwendet. Ein Guard wird einer Transition zugeordnet, wobei diese Transition nur genutzt werden kann, wenn sich sein Ausdruck als wahr erfüllt. Im UML-Metamodell besitzt

das Element Guard ein Attribut vom Typ BooleanExpression. Ein solcher boolscher Ausdruck besteht aus zwei Teilen, der Angabe einer Sprache, in der die Bedingung formuliert ist, und der Bedingung selbst in Form eines Strings. Um nun beispielsweise XPath-Ausdrücke in Guards zu ermöglichen, muss lediglich als Sprache des Ausdrucks XPath angegeben werden. Die Auswertung eines konkreten Ausdrucks erfolgt erst bei der Ausführung durch den Prozessinterpreter. Es ist also nicht erforderlich, das UML-Metamodell hierfür zu erweitern.

7.3.3 Wohlgeformtheitsregeln

Neben den vorgenommenen Erweiterungen des UML-Metamodells gibt es folgende Einschränkungen, die für Prozessdiagramme eingehalten werden müssen, damit sich eine korrekte Semantik ergibt:

1. Zustände im Prozessdiagramm müssen immer mit dem Stereotyp «stateWithPorts» klassifiziert sein, da alle Zustände mit Ports versehen sein müssen (vgl. Abschnitt 6.3.1).

```
context StateVertex inv:
          self.isStereotypedAs(stateWithPorts) or
          self.isStereotypedAs(serviceState)
```

2. Für einfache Zustände (SimpleState) dürfen lediglich CallStates in Verbindung mit dem Stereotyp «serviceState» benutzt werden:

```
context SimpleState inv:
          self.oclIsKindOf(CallState) and self.isStereotypedAs(serviceState)
```

3. Wenn ein StateVertex kein FinalState oder PseudoState ist, muss er genau eine eingehende und eine ausgehende Transition haben. Diese Regel gilt somit für alle SimpleState-, SynchState- und CompositeState-Instanzen; PseudoState-Instanzen wie Startzustand oder Verzweigung sind ausgenommen.

```
context StateVertex inv:
          if not (self.oclIsKindOf(FinalState) or self.oclIsKindOf(PseudoState))
          then (self.incoming->size()=1 and self.outgoing->size()=1)
```

4. Für geschachtelte Aktivitäten in Prozessdiagrammen dürfen lediglich nur wieder Prozessdiagramme benutzt werden:

```
context SubactivityState inv:
          self.submachine.isStereotypedAs(processDiagram)
```

5. Transitionen müssen immer mit dem Stereotyp «typedTransition» klassifiziert sein.

```
context Transition inv:
          self.isStereotypedAs(typedTransition)
```

6. Da auf die Behandlung von Ereignissen verzichtet wurde (siehe Abschnitt 6.2.6), dürfen für Transitionen keine expliziten Trigger-Ereignisse angegeben werden. Dies ist eine Verschärfung der Wohlgeformtheitsregeln 2.12.3.8.[5] und 2.13.3.2.[3] aus der UML-Spezifikation [Omg01], in denen der Gebrauch von Triggern bereits für Aktivitätendiagramme stark eingeschränkt wird.

```
context Transition inv:
          self.trigger->isEmpty()
```

7. Bedingte Verzweigungen und Zusammenführungen (Diamant-Symbol) müssen in *decision* und *merge* unterscheidbar sein.

```
context PseudoState inv:
          (self.kind = #junction) implies
          (((self.incoming->size() = 1) and (self.outgoing->size() > 1)) or
           ((self.outgoing->size() = 1) and (self.incoming->size() > 1)))
```

Ein solcher Knoten hat entweder nur eine eingehende und mehrere ausgehende Transitionen (decision) oder umgekehrt mehrere eingehende aber nur eine ausgehende Transitionen (merge). Eine Verzweigung mit sowohl mehreren eingehenden als auch mehreren ausgehenden Transitionen kann durch die sequentielle Kombination eines merge- und eines decision-Knotens nachgebildet werden.

8. Enthält der in einem Guard angegebene Ausdruck Bezüge auf Elemente im Migrationsdokument, so beziehen sich diese auf das vom Ausgangsport des vorausgegangenen Zustands ausgegebene Dokument, noch bevor es eventuell auf der Transition transformiert wird. Die verwendeten Elemente im Bedingungsausdruck des Guards müssen für die Struktur der durch den Port vorgegebenen Dokumenttypen wohldefiniert sein. So darf ein Guard zum Beispiel nur XML-Elemente enthalten, die in den zu den Dokumenttypen des Ports gehörigen Schemata als Unterelemente definiert sind. Ansonsten ist die Semantik des Guards nicht eindeutig definiert.

Die bei der obigen Definition der Spracherweiterung für OCL-Ausdrücke benutzte Funktion *isStereotypedAs* überprüft, ob ein Element mit dem angegebenen Stereotyp klassifiziert wurde. Die Operation ist folgendermaßen definiert:

```
context ModelElement def:
    let isStereotypedAs(stype:Name):Boolean =
        self.stereotyp->exists(s|s.name=stype)
```

7.4 Implementierung des erweiterten Metamodells

Damit die in den nachfolgenden Kapiteln vorzustellenden Software-Werkzeuge auf Prozessdiagrammen operieren können, muss das im letzten Abschnitt eingeführte Metamodell für Prozessdiagramme implementiert werden. Nötig ist eine Datenstruktur, in der sich beliebige Prozessdiagramme mit ihren statischen und dynamischen Teilen ablegen lassen. Da wegen ihrer Plattformunabhängigkeit für alle Implementierungen dieser Arbeit die Programmiersprache Java gewählt wurde, soll auch das Metamodell für Prozessdiagramme als eine Sammlung von Java-Klassen implementiert werden. Dazu wird im Folgenden beschrieben, wie die Metaklassen des UML-Profils im Einzelnen in Java-Klassen übertragen wurden. Grundsätzlich repräsentiert dabei jede Java-Klasse eine entsprechende Metaklasse. Dadurch entsteht eine effiziente Datenstruktur, die durch ihre Analogie zum Metamodell für einen Kenner des Metamodells leicht zu handhaben und in andere Werkzeuge zu integrieren ist. Außerdem stellt die Anlehnung an das Metamodell sicher, dass sich alle syntaktischen Elemente der Diagramme, aber auch alle Einschränkungen, in der Implementierung des Metamodells wiederfinden, und so eine fehler- und verlustfreie Speicherung von Prozessdiagrammen in dieser Datenstruktur möglich ist. Die Dokumentation der Implementierung erfolgt anhand von UML-Klassendiagrammen. Diese Notation eignet sich als Grundlage für die Implementierung des Metamodells auch in einer anderen objekt-orientierten Sprache als Java. Einschließlich der notwendigen Werkzeuge (siehe Abschnitte 7.5 und 8.4) umfasst die Umsetzung in Java etwa 10.000 LOC (lines of code).

Wie schon erwähnt, besteht das Metamodell für Prozessdiagramme aus dem UML-Metamodell für Aktivitätendiagramme (siehe [Omg01] 2.12) bzw. Zustandsdiagramme (siehe [Omg01] 2.13) und den in Abschnitt 7.3 daran vorgenommenen Erweiterungen und Einschränkungen. Die bereits aus Abschnitt 7.3 bekannte Abbildung 7.6 gibt noch einmal einen Überblick über dieses, den Prozessdiagrammen zu Grunde liegende Metamodell.

Der obere Teil der Abbildung ist ein Auszug aus dem UML-Metamodell (vgl. [Omg01] Abb. 2-24 und 2-30), bei dem bereits einige Klassen ausgelassen wurden, wenn sie für die Darstellung von Prozessdiagrammen nicht relevant sind. Dazu gehören zum Beispiel alle Klassen, die mit der Behandlung von Ereignissen zu tun haben (z.B. „Event“, siehe [Omg01] Abb. 2-24) oder die Zustände beschreiben, die in Prozessdiagrammen nicht vorkommen können (z.B. „ObjectFlowState“, siehe [Omg01] Abb. 2-30).

Im unteren Teil der Abbildung befinden sich die Stereotypen, die zur Erweiterung von Syntax und Semantik der konventionellen Aktivitätendiagramme ergänzt wurden. Nach dem UML-Metamodell befinden sich diese Stereotypen konzeptionell auf der Ebene unterhalb des eigentlichen Metamodells, sie erfüllen jedoch ähnliche Funktionen wie die Metaklassen, da auch sie zur Klassifizierung von Modellelementen benutzt werden sollen. Man spricht in diesem Zusammenhang auch von „virtuellen Metaklassen“ (siehe [Omg01] 2-78).

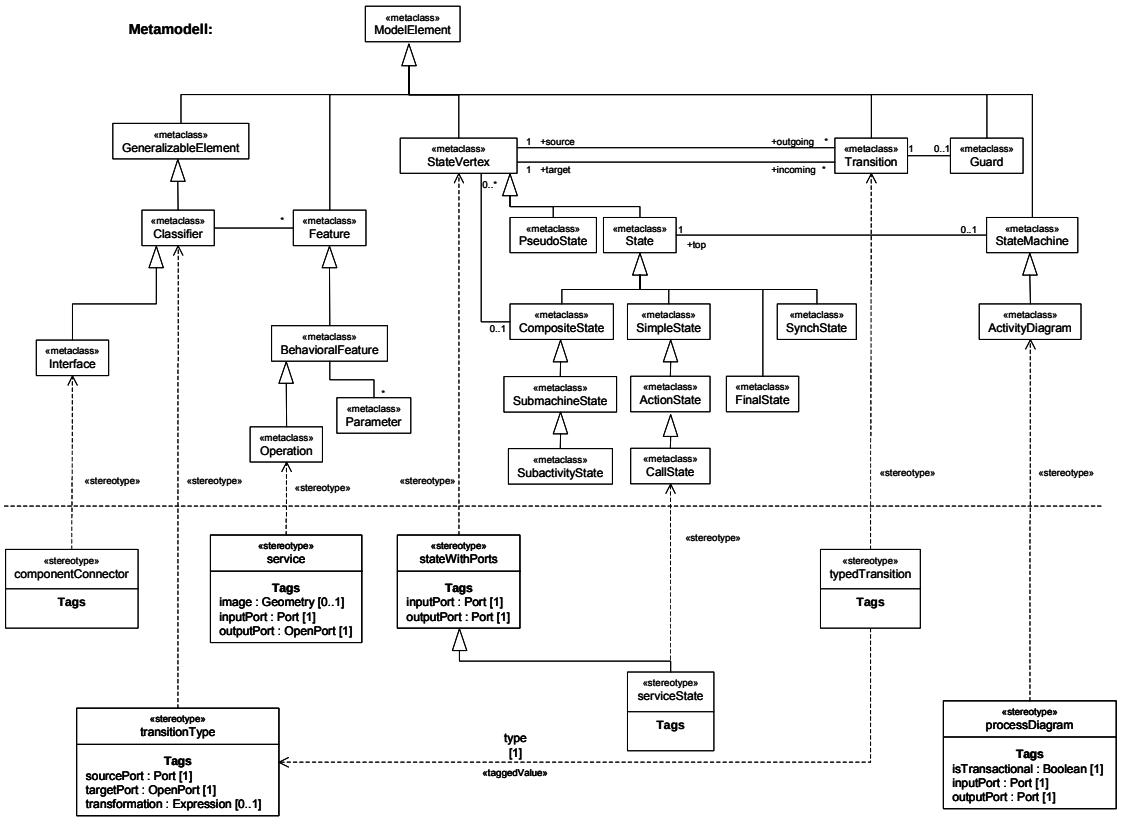


Abb. 7.6: UML-Metamodell für Prozessdiagramme

Da kein allgemeines UML-Metamodell, sondern ein spezielles Metamodell für Prozessdiagramme implementiert werden soll, bietet es sich an, bei der Implementierung die Ebenentrennung aufzuheben und die vorgenommenen Erweiterungen und Stereotypen gleichwertig zu den Original-Metaklassen in der Modellimplementierung für Prozessdiagramme zu integrieren. Zur Verschmelzung der beiden Ebenen kommen zwei Techniken zum Einsatz: Zum einen kann ein Stereotyp dadurch ersetzt werden, dass seine Tag-Definitionen, die neue Meta-Attribute der Basisklasse darstellen sollen, direkt als zusätzliche Attribute dieser Basisklasse hinzugefügt werden. Zum anderen kann man eine Unterklasse der originalen Basisklasse bilden, die dann alle Eigenschaften des Stereotyps mit den geerbten Attributen und Methoden der Basisklasse vereinigt. Zusätzlich zu der Verschmelzung von Metaklassen und Stereotypen wurden noch einige weitere Metaklassen angepasst und für unsere Zwecke optimiert, auf die wir an den entsprechenden Stellen noch jeweils näher eingehen werden. Vielfach wurden zum Beispiel die Vererbungshierarchien bereinigt und überflüssige Vererbungsstufen entfernt.

Im Folgenden werden nun die Klassenstrukturen des implementierten Metamodells für Prozessdiagramme als Klassendiagramme dargestellt. Dabei erfolgt zunächst die Präsentation der elementaren Datentypen, dann der Metaklassen für statische Diagrammelemente und schließlich der Metaklassen für die dynamischen Modellelemente von Prozessdiagrammen. Es wird jeweils das zugehörige Klassendiagramm abgebildet, und danach werden die einzelnen Klassen des Diagramms beschrieben.

Klassendiagramm für die Datentypen

Im Abschnitt 6.3 wurde zur Modellierung von Dokumenttypen in das Konzept der Ports eingeführt und daran anknüpfend in Abschnitt 7.3.1 eine Menge von Datentypen defi-

niert, mit denen sich das Port-Konzept in die Prozessdiagramme einbetten lässt. Das dort in Abbildung 7.5 gezeigte Klassendiagramm dient gleichzeitig als Grundlage für die Implementierung dieser Datentypen. Es wird hier aufgegriffen und lediglich leicht angepasst. Zu den Anpassungen gehört die Vererbungsbeziehung zwischen den einzelnen Datentypen und der abstrakten Oberklasse ModelElement, von der im Weiteren sämtliche Klassen des Metamodells abgeleitet werden.

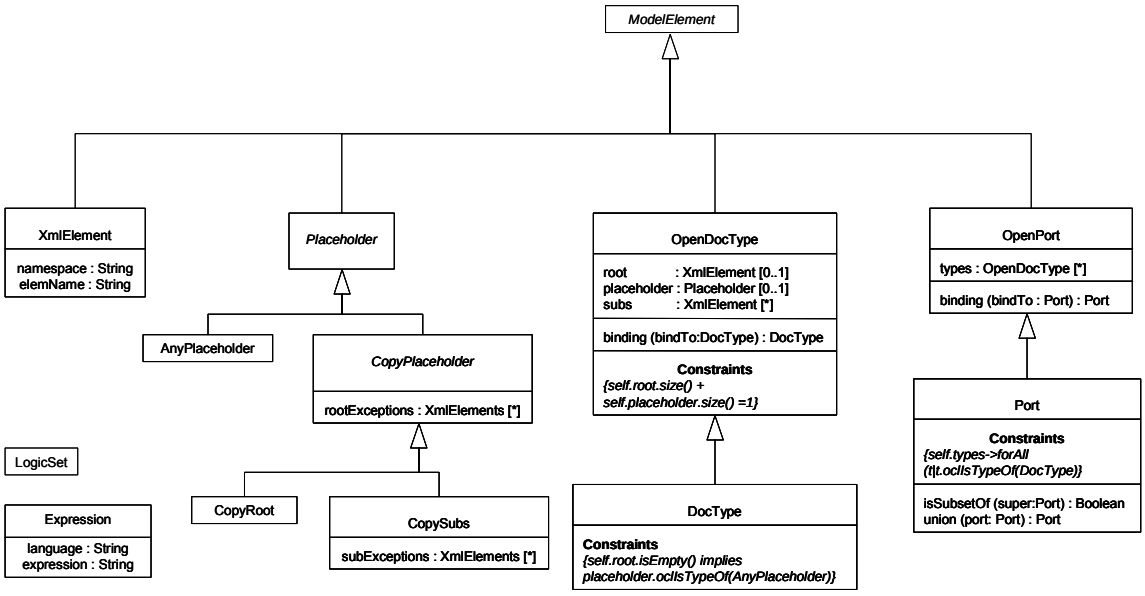


Abb. 7.7: Klassendiagramm für die Datentypen

Da diese Klassen die Funktion von Datentypen übernehmen, werden keine Assoziationen zu diesen Klassen gezogen, sondern sie können einfach als Typen der Attribute anderer Klassen benutzt und im mittleren Fach der jeweiligen Klassensymbole notiert werden. Die spezifizierten Methoden wie `binding(..)` oder `isSubsetOf(..)` müssen entsprechend der zugehörigen Definitionen aus Abschnitt 6.3.3 implementiert werden.

LogicSet

Die Hilfsklasse LogicSet wird benutzt, um eine logische Menge von Elementen abzubilden, die zwei gleiche Elemente jeweils nur einmal enthalten kann. Die Bezeichnung *gleich* bedeutet hier nicht nur *identisch*, sondern schließt auch Objekte ein, die nicht identisch sind, aber den gleichen Typ und gleiche Attributwerte haben. LogicSet kommt bei den Datentypen immer dann intern zum Einsatz, wenn ein Attribut die unbeschränkte Kardinalität (*) besitzt, weil damit laut formaler Definition in Abschnitt 6.3.3 eine logische Menge von Elementen gemeint ist.

Expression

Der Datentyp Expression entspricht dem gleichnamigen UML-Datentyp und erlaubt die Angabe einer Sprache und eines Ausdrucks, der entsprechend der Syntax und Semantik dieser Sprache auszuwerten ist. Expression wird bei Prozessdiagrammen vor allem benutzt, um XML-spezifische Ausdrücke in Sprachen wie XSL oder XPath anzugeben.

Klassendiagramm für das statische Modell

Zum statischen Modell gehören die Konnektoren mit ihren Services und die Transitionstypen. Hinzu kommt die Verwaltung dieser Elemente in Packages (vgl. Abschnitt 8.3.2). Das folgende Klassendiagramm zeigt den Ausschnitt aus dem Metamodell, durch den die statischen Modellelemente realisiert werden sollen.

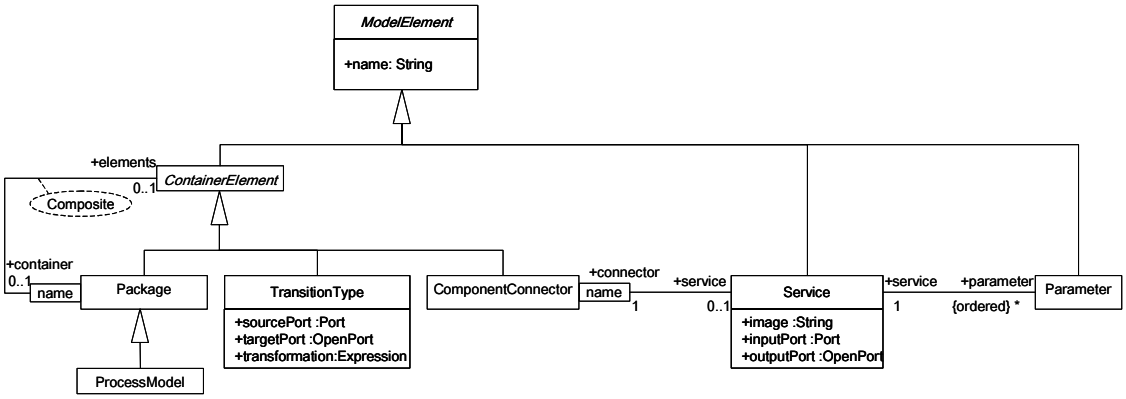


Abb. 7.8: Klassendiagramm für das statische Modell

ModelElement — Alle verwendeten Klassen sind direkt oder indirekt von der gemeinsamen Oberklasse ModelElement abgeleitet. Dadurch erhält jedes Modellelement einen eigenen Namen.

ContainerElement — Die Unterklasse ContainerElement dient als Abstraktion für alle Modellelemente, die in einem Package als Container enthalten sein können.

Package — Ein Package ist ein Container für ContainerElement-Objekte. Dementsprechend hat die Klasse Package eine qualifizierte 1:n-Assoziation zu ContainerElement und kann damit eine Menge beliebiger Objekte enthalten, deren Typ von ContainerElement abgeleitet ist. Da Packages ineinander verschachtelt sein können, ist auch die Klasse Package selbst eine Unterklasse von ContainerElement. Entsprechend dem Entwurfsmuster *Composite* (siehe [Gam96] S.239) kann so eine baumartige Package-Struktur entstehen, an deren Blättern Containerelemente wie Konnektoren oder Transitionstypen hängen.

ProcessModel — Die Klasse ProcessModel als spezielles Package soll als Wurzelknoten dieser Baumstruktur benutzt werden und quasi ein Ursprungspackage darstellen, in dem alle anderen Packages enthalten sind. Damit stände ein Objekt vom Typ ProcessModel an der Spitze des gesamten Modells.

TransitionType — Die Klasse TransitionType realisiert das Stereotyp «transitionType» mit seinen entsprechenden Attributen (für die Bedeutung der einzelnen Attribute siehe Abschnitt 7.3.2). Die

Attributtypen Expression, Port und OpenPort wurden bereits als Datentypen vorgestellt. Die Zuordnung eines Transitionstypen zu Transitionen erfolgt später bei der Betrachtung der dynamischen Modellelemente.

ComponentConnector — Die Klasse ComponentConnector realisiert das Stereotyp «componentConnector» aus Abschnitt 7.3.2, das ein UML-Interface als Konnektor für eine Softwarekomponente auszeichnet. Diese Klasse übernimmt die Rolle der bisherigen Metaklasse Interface. Wie Abbildung 7.6 zeigt, wird Interface von Classifier abgeleitet und erbt dadurch eine Assoziation zur Klasse Feature, die dann durch Vererbung zum Stereotyp «service» führt. Diese Vererbungsstufen konnten eingespart werden und stattdessen direkt eine Assoziation zwischen ComponentConnector und der neuen Klasse Service gezogen werden. Da zunächst nicht vorgesehen ist, Vererbungskonzepte auf Instanzen von ComponentConnector anzuwenden, konnte auch die Ableitung von der Metaklasse GeneralizableElement entfallen.

Service — Die Klasse Service realisiert das Stereotyp «service» aus Abschnitt 7.3.2, mit dem Methoden eines ComponentConnectors ausgezeichnet werden, wenn sie innerhalb von Prozessaktivitäten aufrufbar sein sollen. Diese Klasse übernimmt die Rolle der Metaklasse Operation. Die Vererbungsstufen Feature und BehavioralFeature (siehe Abbildung 7.6) konnten eingespart werden. Dadurch ergibt sich auch die direkte Assoziation zur Metaklasse Parameter. Die Bedeutung der Attribute ergibt sich aus der Stereotyp-Definition in Abschnitt 7.3.2.

Parameter — Die Klasse Parameter entspricht der gleichnamigen UML-Metaklasse. Mit ihr werden die für einen Service-Aufruf nötigen Parameter spezifiziert. Die Klasse hat keine Attribute, weil die Angabe eines eindeutigen Namens bereits durch die Ableitung von ModelElement ermöglicht ist. Die Angabe eines Datentyps für jeden Paramter ist nicht nötig, weil für Prozessdiagramme zunächst nur der Datentyp String vorgesehen ist (vgl. Abschnitt 6.2.2).

Klassendiagramm für das dynamische Modell

Zum dynamischen Modell gehören die einzelnen Prozessdiagramme mit ihren Zuständen und Transitionen (vgl. Abschnitt 6.2). Das folgende Klassendiagramm zeigt den Ausschnitt aus dem Metamodell, durch den die dynamischen Modellelemente realisiert werden sollen.

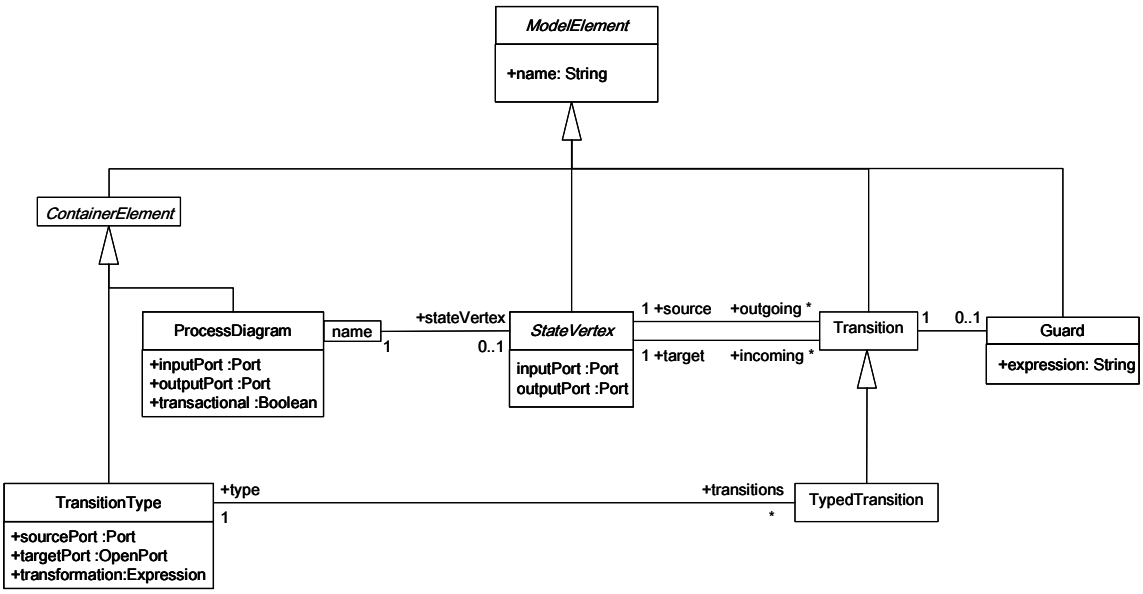


Abb. 7.9: Klassendiagramm für das dynamische Modell

ProcessDiagram

Die Klasse ProcessDiagram realisiert das Stereotyp «processDiagram» aus Abschnitt 7.3.2, mit dem UML-Aktivitätendiagramme als Prozessdiagramme ausgezeichnet werden. Diese Klasse übernimmt damit die Rolle der Metaklasse ActivityDiagram. Die Vererbungsstufe StateMachine (siehe Abbildung 7.6) konnte eingespart werden. Im UML-Metamodell hat die Klasse StateMachine eine 1:n-Assoziation zu den Transitionen des Zustandsdiagramms (siehe [Omg01] Abb. 2-24). Uns erschien es dagegen zweckmäßiger, eine solche Assoziation zur Klasse StateVertex anzulegen, um so die Verbindung zu allen Zuständen des Prozessdiagramms herzustellen. Es handelt sich dabei um eine qualifizierte Assoziation, wodurch sichergestellt ist, dass jeder Zustand innerhalb eines Prozessdiagramms einen eindeutigen Identifikator als Namen besitzt. Für die Bedeutung der Attribute sei wiederum auf die Definition des Stereotypen «processDiagram» in Abschnitt 7.3.2 verwiesen.

StateVertex

Die Klasse StateVertex realisiert das Stereotyp «stateWithPorts» aus Abschnitt 7.3.2, mit dem alle Zustände eines Prozessdiagramms ausgezeichnet werden müssen. Diese Klasse übernimmt gleichzeitig die Rolle der gleichnamigen UML-Metaklasse. Damit handelt es sich auch hier um eine Verschmelzung von Metaklasse und Stereotyp. Die Bedeutung der Attribute inputPort und outputPort wurde in Abschnitt 7.3.2 bei der Stereotyp-Definition erläutert. Jeder StateVertex kann über die zugehörigen Assoziationen ein- und ausgehende Transitionen haben.

Transition — Die Klasse Transition entspricht der gleichnamigen UML-Metaklasse aus Abbildung 7.6. Jede Transition verbindet zwei Zustände (source und target) miteinander.

Guard — Die Klasse Guard entspricht der gleichnamigen UML-Metaklasse aus Abbildung 7.6. Ein Guard ist immer einer Transition zugeordnet. In dem Attribut expression kann die Transitionsbedingung angegeben werden.

TypedTransition — Die Klasse TypedTransition realisiert das Stereotyp «typedTransition» aus Abschnitt 7.3.2, mit dem alle Transitionen eines Prozessdiagramms ausgezeichnet werden müssen. Sie erbt alle Eigenschaften der Oberklasse Transition und erhält zusätzlich eine Assoziation zu TransitionType, mit der jedem Transitions-Objekt ein Typ aus dem statischen Modell zugewiesen werden kann. Diese Assoziation koppelt somit für den Bereich der Transitionen den statischen mit dem dynamischen Modellteil.

Klassendiagramm für die verschiedenen Zustandsarten

In Prozessdiagrammen können verschiedene Arten von Zuständen vorkommen. Aus diesem Grund muss die allgemeine Abstraktionsklasse StateVertex aus dem Klassendiagramm für dynamische Modellelemente (siehe Abbildung 7.9) weiter spezialisiert werden. Das folgende Klassendiagramm zeigt sämtliche Unterklassen von StateVertex und gegebenenfalls ihren Zusammenhang mit den anderen Klassen des Metamodells.

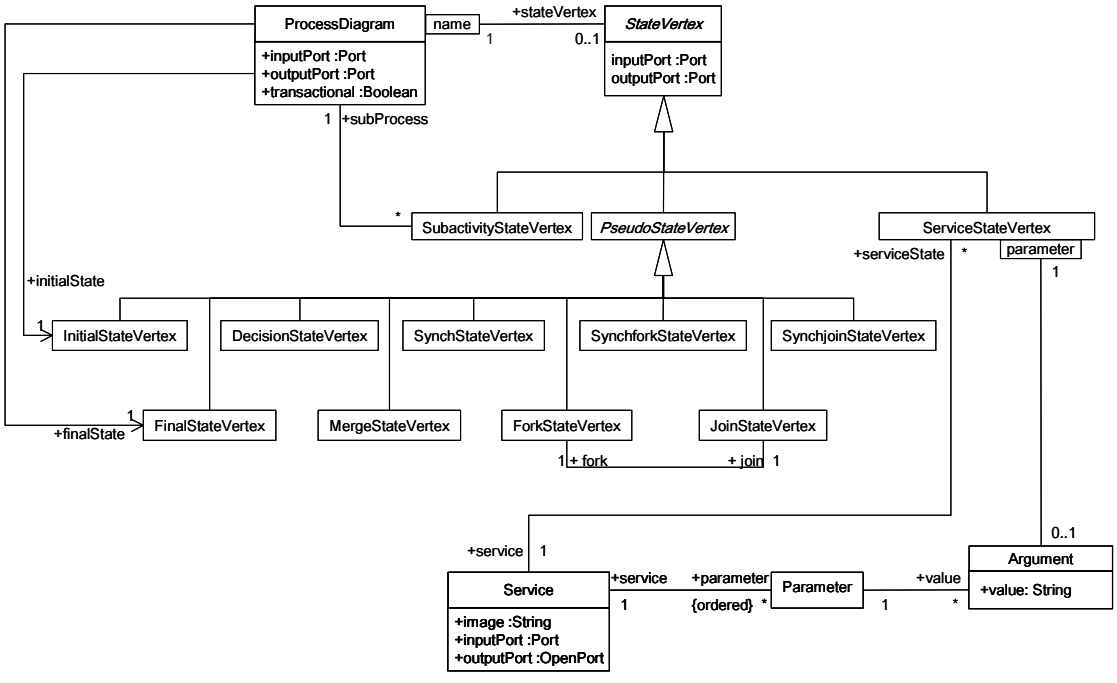


Abb. 7.10: Klassendiagramm für verschiedene Zustandsarten

SubactivityStateVertex Die Klasse SubactivityStateVertex übernimmt die Rolle der UML-Metaklasse SubactivityState (siehe Abbildung 7.6), durch die ein Subprozess in ein Prozessdiagramm eingebettet werden kann; daher auch die Assoziation zur Klasse ProcessDiagram.

ServiceStateVertex Die Klasse ServiceStateVertex vereinigt die Eigenschaften des Stereotyps «serviceState» aus Abschnitt 7.3.2 und dessen Basisklasse CallState aus dem UML-Metamodell (siehe Abbildung 7.6). Im UML-Metamodell ist die Klasse CallState von State abgeleitet, die wiederum über eine Assoziation mit einer Action verbunden ist, bei der es sich um eine CallAction handeln muss, mit der eine bestimmte Operation zur Ausführung ausgewählt wird (siehe [Omg01] 2.13.3.3). Dieser Sachverhalt wurde im Metamodell für Prozessdiagramme vereinfacht durch die Assoziation zur Klasse Service abgebildet. Somit ist jedem ServiceStateVertex genau ein Service zugeordnet, der bei der Ausführung dieses Zustands ausgeführt werden soll. Damit die Ausführung wohldefiniert ist, wird außerdem für jeden Parameter des Service ein Objekt der Klasse Argument angelegt und mit dem ServiceStateVertex verbunden, in dem der Wert für den jeweiligen Parameter gesetzt werden kann.

Argument Diese Klasse übernimmt die Rolle der gleichnamigen UML-Metaklasse (siehe [Omg01] Abb. 2-15), mit der für den Aufruf einer Operation Werte für deren Parameter angegeben werden.

PseudoStateVertex Die abstrakte Klasse PseudoStateVertex übernimmt ähnlich der UML-Metaklasse PseudoState die Funktion einer Abstraktion aller sonstigen Zustände, in denen keine Aktionen ausgeführt werden müssen. Im Gegensatz zum orginalen UML-Metamodell werden die verschiedenen Unterarten jedoch nicht durch ein Attribut kind (siehe [Omg01] 2.12.2.7) unterschieden, sondern durch die Bildung weiterer Unterklassen, die im Folgenden kurz erläutert werden.

InitialStateVertex Diese Klasse repräsentiert den Startzustand eines Prozessdiagramms. Zwecks effizienteren Zugriffs gibt es eine unidirektionale Referenz von ProcessDiagram auf ein Objekt der Klasse.

FinalStateVertex Diese Klasse repräsentiert den Endzustand eines Prozessdiagramms. Zwecks effizienteren Zugriffs gibt es eine unidirektionale Referenz von ProcessDiagram auf ein Objekt der Klasse.

DecisionStateVertex Diese Klasse repräsentiert eine Oder-Verzweigung.

MergeStateVertex Diese Klasse repräsentiert eine Oder-Zusammenführung.

SynchStateVertex	Diese Klasse repräsentiert einen Synchronisationszustand (vgl. [Omg01] 2.12.2.15) und entspricht der UML-Metaklasse SynchState.
SynchforkStateVertex	Diese Klasse repräsentiert eine Und-Verzweigung vor einem Synchronisationszustand (vgl. Abschnitt 8.3.3).
SynchjoinStateVertex	Diese Klasse repräsentiert eine Und-Zusammenführung nach einem Synchronisationszustand (vgl. Abschnitt 8.3.3).
ForkStateVertex	Diese Klasse repräsentiert eine Und-Verzweigung. Die Assoziation zum zugehörigen JoinStateVertex dient dem effizienteren Zugriff auf die korrespondierende Zusammenführung.
JoinStateVertex	Diese Klasse repräsentiert eine Und-Zusammenführung. Die Assoziation zum zugehörigen ForkStateVertex dient dem effizienteren Zugriff auf die korrespondierende Verzweigung.

Flexible Zugriffsschicht auf Elemente des Datenmodells

Das soeben vorgestellte Datenmodell erlaubt eine einfache Implementierung von Prozessdiagrammen in objekt-orientierten Programmiersprachen. Im Rahmen der Diplomarbeit wurde das Modell mit Java umgesetzt. Um eine möglichst hohe Wiederverwendbarkeit zu gewährleisten, bietet es sich an, diese Implementierung und die zugehörigen Klassen sowohl für das in Kapitel 9 vorgestellte Modellierungswerkzeug, als auch für weitere Werkzeuge wie den Prozessvalidierer (siehe Abschnitt 7.4.2) und den Prozessinterpreter (siehe Abschnitt 10.2.1) zu verwenden. Diese Werkzeuge erfordern jedoch teilweise unterschiedliche Sichten auf die jeweiligen Modellelemente oder zusätzliche Attribute für die einzelnen Klassen. Der Editor zum Beispiel muss für jedes visuelle Modellelement die Koordinaten bezüglich der Zeichenfläche speichern, bei denen ein Symbol für das Modellelement platziert worden ist. Um die Modellelemente dafür mit einer flexiblen und erweiterbaren Schnittstelle auszustatten, wurde das Entwurfsmuster *Adapter* (vgl. [Gam96] S.171) eingesetzt und jedem Modellelement über eine Assoziation eine Adapterklasse zugeordnet. Damit für verschiedene Modellelemente auch verschiedene Adapterarten ermöglicht werden, können spezialisierte Unterklassen des abstrakten Adapters AbstractModelElementAdapter gebildet werden. Auf diese Weise entstehen zwei Vererbungshierarchien, die an der Spitze durch eine Assoziation miteinander verbunden sind. Die folgende Abbildung soll den Einsatz dieses Entwurfsmusters noch einmal verdeutlichen:

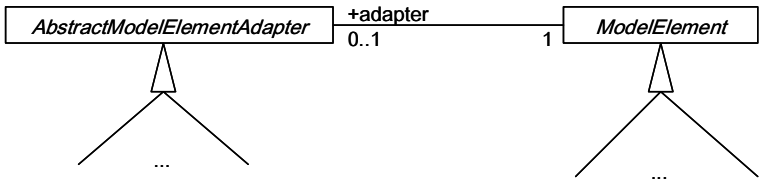


Abb. 7.11: Entwurfsmuster Adapter für Modellelemente

7.5 Validierung von Prozessdiagrammen

Im vorherigen Abschnitt wurde eine Datenstruktur vorgeschlagen, mit der sich Prozessdiagramme in einem implementierten System repräsentieren lassen. Wenn eine Objektstruktur dieser Datenklassen vorliegt, stellt sich die Frage, ob sie ein gültiges Prozessdiagramm widerspiegelt, d.h. ob alle im Abschnitt 7.3 für die Stereotypen aufgestellten Bedingungen erfüllt sind und zudem alle Wohlgeformtheitsregeln eingehalten werden, die bereits in der UML-Spezifikation für Aktivitätendiagramme gefordert sind. Um diese Frage zu beantworten, haben wir einen Validierungsalgorithmus entwickelt, der eine Objektstruktur der obigen Datenklassen als Eingabe erhält und prüft, ob alle Anforderungen erfüllt sind. Ein entsprechendes Werkzeug - im Folgenden *Validierer* genannt - kann an verschiedenen Stellen zum Einsatz kommen; in dem von uns entwickelten System wird es an zwei Stellen verwendet: Zum einen kann der Validierer vom Prozesseditor aus jederzeit aufgerufen werden, um ein beliebiges Prozessdiagramm auf Korrektheit zu prüfen (siehe Abschnitt 9.2.5, Validieren von Prozessdiagrammen). Dies ist besonders während der Modellierung eines Workflow hilfreich, da der Modellierer sofort erkennen kann, an welchen Stellen sein Modell noch unvollständig oder fehlerhaft ist. Zum anderen kann der Validierer wahlweise aufgerufen werden, bevor der Prozessinterpreter (siehe Kapitel 10) einen Prozess ausführt. Dies bietet den Vorteil, dass vorhandene Fehler in einer Prozessbeschreibung so früh wie möglich erkannt werden, d.h. bevor der Interpreter überhaupt mit der Ausführung beginnt.

An dieser Stelle soll der von uns entwickelte Validierungsalgorithmus erläutert werden. Er ist vom Konzept her nicht an die konkrete Implementierung aus Abschnitt 7.4 gebunden und ließe sich daher auch für andere Datenstrukturen realisieren. Da ein Prozessdiagramm einen gerichteten Graph darstellt, führt der Algorithmus eine Tiefensuche ausgehend vom Startzustand des Prozesses durch. Auf diese Weise besucht er Schritt für Schritt alle Zustände. Für jeden gefundenen Zustand werden, abhängig von seinem Typ, alle notwendigen Überprüfungen durchgeführt (vgl. Abschnitt 7.3). Dazu zählt beispielsweise die korrekte Anzahl ein- und ausgehender Transitionen, die Kompatibilität der Ports usw. Für Transitionen wird geprüft, ob Guards korrekt verwendet wurden und die Ports der Transitionstypen kompatibel sind. Während der Tiefensuche werden auch die im Prozess verwendeten Bestandteile des statischen Modells, d.h. Konnektoren und Transitionstypen validiert.

Ein wichtiger Gegenstand der Validierung ist die korrekte Schachtelung von parallelen Prozessabschnitten im Zusammenhang mit Fork- und Join-Zuständen, die den Wohlgeformtheitsregeln der UML-Spezifikation ([Omg01] 2-157f und 2-183f) genügen muss. Dazu zählt beispielsweise die Bedingung, dass alle Pfade, die von einem Fork-Zustand ausgehen, später wieder in einem Join-Zustand vereinigt werden. Zusätzlich dürfen keine Transitionen zwischen verschiedenen dieser Pfade verlaufen, es sei denn, sie führen zu oder aus einem Synchronisationszustand. Im Folgenden soll erläutert werden, wie der Validierer die Einhaltung dieser Regeln prüft. Der Algorithmus verwendet dazu „Farben“, mit denen er die Prozesszustände markiert. Alle Zustände sind zunächst weiß. Anschließend wird eine erste Farbe ungleich weiß gewählt, mit der die Zustände gefärbt werden, wenn sie von der Tiefensuche zum ersten Mal erreicht werden. Falls die Tiefensuche auf einen Fork-Zustand trifft, wählt der Algorithmus für jeden ausgehenden Pfad eine eigene Farbe, die ungleich aller bisher gewählten ist, mit der im Folgenden alle Zustände des Pfades markiert werden. Auf diese Weise lässt sich

für jeden Zustand ermitteln, zu welchem parallelen Pfad er gehört. Alle Transitionen, die im weiteren Verlauf der Tiefensuche gefunden werden und die nicht zu einem Synchronisationszustand führen, dürfen nun ausschließlich zu weißen Zuständen führen oder solchen, die die gleiche Farbe besitzen wie der Zustand, aus dem die Transition herausführt. Durch diese Bedingung wird geprüft, ob Transitionen zwischen verschiedenen Strängen verlaufen, was nicht erlaubt ist. Falls dieser Fall eintritt, liegt eine ungültige Schachtelung vor und es wird eine entsprechende Fehlermeldung ausgegeben. Abbildung 7.12 verdeutlicht den Einsatz der Farben. Die Zahlen geben dabei an, in welcher Reihenfolge die Zustände von der Tiefensuche besucht wurden. Momentan wird Knoten 9 überprüft. Sowohl die Transition zum Zustand 7 als auch zum weißen Zustand stellt kein Problem dar. Die Transition zum Zustand 4 führt jedoch zu einem Zustand, der nicht die gleiche Farbe besitzt und somit nicht zum gleichen parallelen Abschnitt gehört, so dass diese Konstruktion nicht wohlgeformt ist.

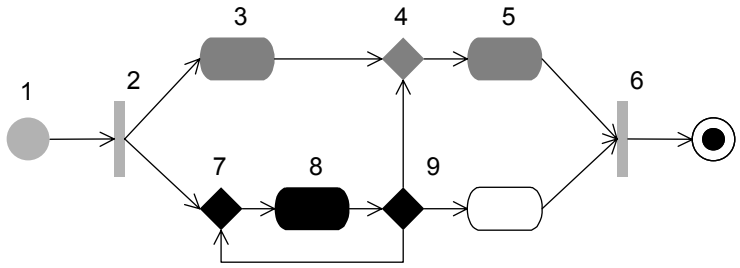


Abb. 7.12: Verwendung von Farben im Validierungsalgorithmus

In Abbildung 7.13 ist der Validierungsalgorithmus in Auszügen abgebildet. Dabei sind alle Anweisungen für Prüfungen von Ports, Anzahl der Transitionen usw. nicht dargestellt, da sie zum Teil sehr technisch sind und nicht zum allgemeinen Verständnis des Algorithmus beitragen. Stattdessen soll der Ablauf der Tiefensuche und das Verfahren des Färbens von Zuständen verdeutlicht werden. Es handelt sich um einen rekursiven Algorithmus, in dem der Begriff „Section“ verwendet wird. Darunter verstehen wir einen Prozessabschnitt, der aus einem Zustand und einem Rest-Abschnitt besteht. Ein gesamtes Prozessdiagramm besteht demzufolge aus dem Startzustand und einem Rest, welcher alle übrigen Zustände umfasst.

```
validateProcess(process):
1  states := process.getAllStates()
2  for i:=1 to states.length do
      /* Alle Zustände weiß färben */
3     state := states[i]
4     color[state] := WHITE
5  end for
6  colorOfSection := determine color ≠ WHITE
7  validateSection(process.getInitialState(), color, colorOfSection)

validateSection(state, color, colorOfSection):
8  c := color[state]
9  if c=WHITE then
10    color[state] := colorOfSection
11    /* führe zustandsabhängige Prüfungen durch */
12    ...
```

```
13    validateSuccessorStates(state, color, colorOfSection)
14 else if c=colorOfSection
15    /* korrekte Schachtelung; Knoten wurde bereits validiert */
16 else /* ungültige Schachtelung */
17    createErrorMessage(...)
18 end if

validateSuccessorStates(state, color, colorOfSection):
19 if state is ForkState
20    states := state.getAllSuccessors()
      /* validiere parallele Stränge */
21    for i:=1 to states.length do
22          c := determine new unique color
23          validateSection(states[i], color, c)
24    end for
25 else if state is SynchState
26    ...
27 ...
28 else /* Knoten mit genau einer Ausgangstransition */
29    validateSection(state.getSuccessor(), color, colorOfSection)
30 end if
```

Abb. 7.13: Validierungsalgorithmus

7.6 Automatische Portberechnung

Bei der Modellierung von Prozessdiagrammen ist eine umfangreiche Menge von Ports zu spezifizieren, um damit die Dokumentstrukturen der Workflows vollständig zu modellieren: Jeder Service und jeder Transitionstyp im statischen Modell muss mit einem Ein- und einem Ausgangsport versehen werden. Genauso bedarf im dynamischen Modell jedes Prozessdiagramm als Ganzes und ebenso jeder einzelne Zustand der Angabe eines Ein- und eines Ausgangsports (vgl. Abschnitte 6.3 und 7.3). Um ein vollständig konsistentes Workflow-Modell aufzustellen, müsste der Modellierer also einen nicht unerheblichen Aufwand mit der Angabe aller nötigen Ports treiben. Allerdings erlauben die in Abschnitt 7.3 bei der formalen Definition der Sprache für Prozessdiagramme gemachten Einschränkungen keinen großen Spielraum für die Ausgestaltung von Ports im dynamischen Modell. Für viele Zustände ergeben sich bei Berücksichtigung der Einschränkungen und Constraints die Ein- und Ausgangsports bereits teilweise oder sogar vollständig aus den Vorgaben der Sprachdefinition oder dem Kontext innerhalb des Prozessdiagramms. Beispielhaft seien hier die PseudoStates genannt, bei denen häufig der unbeschränkte Port #any[] als Eingangsport gesetzt werden kann.

In sehr vielen Fällen ist somit eine automatische Berechnung der Ports möglich, die zum Ziel hat, zum einen den Benutzer zu entlasten und ihm die automatisierbaren Aufgaben abzunehmen, zum anderen gleichzeitig inkonsistente und fehlerhafte Eingaben des Benutzers zu verhindern. Dabei soll die Angabe eines Ports durch den Benutzer nur noch bei den Elementen des statischen Modells und beim Eingangsport von Prozessdiagrammen möglich sein. Alle anderen Ports ergeben sich im Zusammenhang mit den modellierten Prozessen aus diesen Benutzervorgaben. Insbesondere die Angabe von Ports für die einzelnen Zustände im Prozessdiagramm bleibt dem Benutzer erspart. Lediglich bei den Service-Aktivitätszuständen kann der Benutzer noch Anpassungen

gemäß Abschnitt 6.3.1 vornehmen. Der folgende Abschnitt erklärt die Realisierung der automatischen Portberechnung anhand des dafür implementierten Algorithmus.

Der Algorithmus kann für den Benutzer transparent von einem Modellierungswerkzeug aufgerufen und so das gerade bearbeitete Prozessdiagramm ständig in einen bezüglich der Ports konsistenten Zustand gebracht werden (vgl. Abschnitt 9.1). Da beim Einsatz offener Ports (vgl. Abschnitt 6.3.1) und deren Bindung sowie bei den Ports von Zusammenführungszuständen wie merge und join das Ergebnis der Portberechnung vom Kontext abhängt, in dem sich die Zustände befinden, und sich die Änderung eines Ausgangsports sukzessive auf nachfolgende Zustände auswirken kann, sollte die Portberechnung jeweils neu angestoßen werden, wenn sich der Diagrammkontext durch Hinzufügen oder Löschen einer Transition geändert hat. Eine solche Neuberechnung ist auch ratsam, wenn der Benutzer Portvorgaben im statischen Modell ändert, da dies immer auch Auswirkungen auf die Ports in denjenigen Workflow-Modellen hat, in denen diese statischen Elemente wie Services und Transitionstypen benutzt werden.

Im Folgenden wird das Vorgehen bei der Portberechnung etwas näher skizziert. Dabei wird zwar davon ausgegangen, dass ein Prozessdiagramm in Form der Datenstruktur aus Abschnitt 7.4 vorliegt, die Konzepte ließen sich aber ohne Weiteres auch für andere Implementierungen umsetzen. Zunächst erhält die Klasse StateVertex aus diesem Datenmodell die beiden abstrakten Methoden `calculateInputPort()` und `calculateOutputPort()`, die von allen Unterklassen implementiert werden müssen und die einen Ein- bzw. Ausgangsport den formalen Vorgaben entsprechend berechnen und zurückgeben. Dem objekt-orientierten Konzept der dynamischen Methodenbindung folgend, wird bei Aufruf dieser Methoden die Implementierung der jeweiligen Unterklasse aufgerufen. So können die verschiedenen Unterklassen von StateVertex je nach ihrem Typ die passende Bedingung zum Beispiel aus den Constraints des Stereotyps «stateWithPorts» (siehe Abschnitt 7.3.2) umsetzen. Beispielhaft soll die Klasse MergeStateVertex betrachtet werden: Gemäß den Bestimmungen aus 7.3.2 für Oder-Zusammenführungen (merge) liefert die Methode `calculateInputPort()` für den Eingangsport hier stets den unbeschränkten Port #any[], `calculateOutputPort()` für den Ausgangsport die Vereinigung aller eingehenden Ports zurück.

Die Implementierung der beiden Methoden ist eine grundlegende aber noch nicht hinreichende Voraussetzung für die Durchführung der automatischen Portberechnung. Es reicht nämlich nicht aus, für jeden Zustand eines Prozessdiagramms einmal die beiden Methoden aufzurufen, sondern die Aufrufe müssen vielmehr in einer bestimmten Reihenfolge vorgenommen werden, weil der Ausgangsport vieler Zustände von den Ports der Vorgängerzustände abhängt. Aus diesem Grund werden zunächst die Ports des Startzustands berechnet, weil dieser keine Vorgänger hat, und dann sukzessive durch Breitensuche alle anderen Zustände abgearbeitet. Da in der Modellierungsphase das Diagramm unter Umständen noch nicht vollständig erstellt ist, kann es neben dem Startzustand noch andere Zustände geben, die ebenfalls keine eingehenden Transitionen besitzen und daher bei der Breitensuche so nicht erreicht würden. Darum beginnt die Breitensuche nicht nur beim Startzustand, sondern bei allen Zuständen, deren Eingangsgrad gleich null ist (siehe Zeile 1 des folgenden Pseudocodes). Die folgende Abbildung zeigt den implementierten Algorithmus in vereinfachter Form als Pseudocode.

```
propagatePorts():
 1 queueOfStates := processDiagram.getIndegree_0_States()
 2 setOfVisited := {}
 3 while not queueOfStates.isEmpty()
 4    state            := queueOfStates.dequeue()
 5    state.inputPort  := state.calculateInputPort()
 6    state.outputPort := state.calculateOutputPort()
 7    setOfVisited     := setOfVisited ∪ {state}
 8    listOfSuccessors := state.getSuccessors()
 9    while listOfSuccessors.hasMoreElements()
10       successor  := listOfSuccessor.nextElement()
11       if ((state.outputPort has been modified) or
12           (successor not in setofVisited)) and
13           (successor not in queueOfStates)
14       then queueOfStates.enqueue(successor);
15    end while
16 end while
```

Abb. 7.14: Algorithmus für die automatische Portberechnung

In der Schlange queueOfStates werden die noch zu bearbeitenden Zustände des Prozessdiagramms gespeichert. Solange diese Schlange nicht leer ist, werden ihre Elemente sukzessive abgearbeitet (siehe Zeilen 3 und 4). Dazu wird für den jeweils der Schlange entnommenen Zustand der Ein- und der Ausgangsport neu berechnet (Zeilen 5 und 6). Der besuchte Knoten wird dann durch Einfügen in die Menge setOfVisited markiert (siehe Zeile 7). Anschließend wird die Breitensuche fortgesetzt, indem alle Nachfolgezustände betrachtet werden (Zeilen 9 und 10). Wenn bei dem gerade besuchten Zustand der Ausgangsport durch `calculateOutputPort()` geändert wurde, werden alle Nachfolgezustände in die Schlange zur weiteren Bearbeitung aufgenommen (Zeile 11), ansonsten nur diejenigen, die noch nicht besucht worden sind (Zeile 12). Zustände, die sich jedoch schon in der Schlange befinden, brauchen nicht erneut aufgenommen zu werden (Zeile 13).

Durch die Bedingung in Zeile 11 ist es möglich, dass schon besuchte Zustände erneut in die Warteschlange aufgenommen werden. Aus diesem Grund muss an dieser Stelle eine Bemerkung zur Terminierung des Algorithmus gemacht werden, denn Zyklen im Prozessdiagramm könnten ja eventuell zu Endlosschleifen führen und eine Terminierung verhindern. Die folgende Abbildung zeigt ein schematisches Beispiel für Zyklen im Prozessdiagramm. Bei der Breitensuche würde zunächst die Zusammenführung A und später die Verzweigung B bearbeitet werden, was wiederum Auswirkungen auf A als Nachfolger von B haben kann. Ändert sich dabei der Ausgangsport von A, wird der Zyklus erneut durchlaufen.

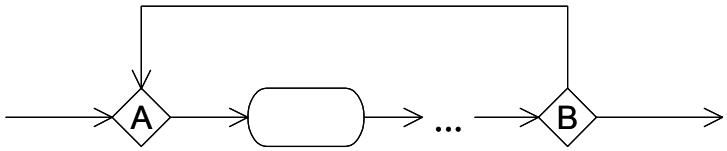


Abb. 7.15: Zyklen bei der automatischen Portberechnung

Bei der Betrachtung der verschiedenen Zustandsarten fällt auf, dass der Ausgangsport eines Zustands nicht kleiner werden kann, wenn die vorgelagerten Ports bezüglich der Anzahl ihrer Dokumenttypen größer werden. Angenommen, bei jedem Zyklendurchlauf würden sich die Ausgangsports immer wieder ändern, was für eine Endlosschleife der Fall sein müsste. Dann könnte sich der Ausgangsport von Zustand A höchstens jedes Mal vergrößern, da er alle eingehenden Ports vereinigt. Da nun aber innerhalb der Ports, die im Zyklus berechnet werden, nur endlich viele XML-Elemente verwendet werden, gibt es auch nur endlich viele Möglichkeiten, diese zu Dokumenttypen zu kombinieren, so dass spätestens dann die Bearbeitung der Schleife abgebrochen wird, wenn der maximale Port mit allen möglichen Dokumenttypen im Zustand A erreicht wurde. Somit terminiert die Breitensuche und damit der Algorithmus zur automatischen Portberechnung nach einer endlichen Anzahl von Rechenschritten.

8 Beschreibung von Prozessen in XML

In den vorausgegangenen Kapiteln haben wir Prozessdiagramme vorgestellt und definiert. Sie bieten eine adäquate Möglichkeit, Workflows entsprechend den Anforderungen aus Kapitel 3 zu modellieren. Bei der Anforderungsanalyse wurde außerdem von einem Workflow-Management-System gefordert, dass es das grafische Workflow-Modell in eine textuell kodierte Form bzw. ein XML-Format übersetzen kann, mit der das Modell ebenfalls vollständig beschrieben wird. Zweck ist dabei, diese Beschreibung als Eingabe für einen Interpreter zu benutzen, der Instanzen des Workflows ausführen soll.

Dieses Kapitel stellt zunächst die Ziele und Anforderungen an eine solche Beschreibungssprache dar und evaluiert in einem zweiten Abschnitt vorhandene Ansätze und Lösungen gegen diese Anforderungen. Nachdem die Stärken und Schwächen der betrachteten Formate aufgezeigt wurden, stellt der dritte Unterabschnitt die Process Markup Language (PML) als unsere Antwort auf die Anforderungen vor. Dabei wird auch der Zusammenhang zwischen dem grafischen UML-Modell der Prozessdiagramme und dem XML-Format erläutert. Diese Abbildung zwischen UML und PML soll bei der Implementierung als Grundlage dienen, um ein Diagramm als PML-Dokument abzulegen bzw. umgekehrt ein PML-Dokument als Prozessdiagramm zu visualisieren. Zur praktischen Arbeit mit PML wurden diese Abbildungsvorschriften umgesetzt und entsprechene Werkzeuge implementiert, die ein PML-Dokument einlesen und die dort gespeicherten Prozessmodelle in der Datenstruktur für das Metamodell (siehe Abschnitt 7.4) ablegen (PML-Importer) bzw. eine Instanz des implementierten Metamodells als PML-Dokument abspeichern können (PML-Exporter). Abschnitt 8.4 beschreibt Aufbau und Arbeitsweise dieser nützlichen Werkzeuge.

8.1 Anforderungen an Prozessbeschreibungssprachen

Als Maßstab für die sich anschließende Evaluierung vorhandener XML-Beschreibungsformate für Workflow-Modelle und als Richtlinie für den Entwurf von PML wurden die in diesem Abschnitt aufgeführten Anforderungen aufgestellt. Die einzelnen Punkte ergaben sich dabei aus den Zielsetzungen des Workflow-Management-Systems, den Eigenschaften von Prozessdiagrammen oder einer besonders dargelegten Motivation. Sie gliedern sich in Muss-Bestimmungen (Punkte 1 bis 10) und Soll-Bestimmungen (Punkte 11 bis 13). Eine Beschreibungssprache eignet sich demnach nur dann für die Beschreibung von Prozessdiagrammen, wenn sie mindestens alle Muss-Bedingungen erfüllt.

1. XML-Format: Eine Prozessbeschreibung sollte als XML-Dokument abgelegt werden. Für die Forderung nach einem XML-Format als Kodierung der Workflow-Beschreibung gibt es mehrere Gründe: XML bietet sich aus Homogenitätsgründen an, weil das Workflow-Management-System insgesamt für die Verarbeitung von XML-Dokumenten ausgelegt ist und demnach wichtige Werkzeuge für die Analyse, Trans-

formation und andere Operationen auf den Dokumenten bereits enthalten muss. Außerdem ist XML ein plattformneutrales Format, was besonders bei heterogenen E-Business-Plattformen relevant ist und die Plattformunabhängigkeit des gesamten WFMS unterstützt. Als weit verbreiteter Standard ist XML zudem kompatibel zu den gängigen Kommunikationsprotokollen des Internets und damit besonders gut für den unternehmensübergreifenden Austausch geeignet. Damit wird der Grundstein für eine gegenseitige Übermittlung von Prozessdefinitionen zwischen Geschäftspartnern gelegt. Es ist zum Beispiel denkbar, dass mit dem eingehenden Migrationsdokument nicht ein vorhandener Prozess des WFMS aufgerufen wird, sondern dass die Prozessbeschreibung selbst ein Bestandteil des Dokuments ist und vom WFMS interpretiert wird. Damit könnten Klienten eigene, individuelle Prozesse definieren und als XML-Dokument zusammen mit der Dateneingabe an das WFMS übermitteln. Als weiterer Vorteil sei noch erwähnt, dass sich die XML-Dokumente mit vorhandenen Transformationskonzepten leicht in ein anderes Datenformat konvertieren lassen. So wäre mit einem passenden XSL-Stylesheet zum Beispiel die Generierung von Programmcode in einer beliebigen Programmiersprache denkbar, mit dem sich das Modell ebenfalls ausführen lassen könnte. Alle diese Gründe haben dazu geführt, dass in letzter Zeit ein Trend zur Speicherung von Prozessmodellen in XML-Formaten zu verzeichnen ist; einige der bestehenden Ansätze sollen im nachfolgenden Abschnitt näher vorgestellt werden.

2. Graphorientierte Beschreibung des Kontrollflusses: Prozessdiagramme als spezielle UML-Aktivitätendiagramme beruhen auf den Prinzipien einer Zustandsmaschine bzw. eines endlichen Automaten (vgl. [Boo99] S.294). Somit handelt es sich bei Prozessdiagrammen im Wesentlichen um gerichtete Graphen mit den Zuständen und Aktivitäten als Knoten und den Transitionen als Kanten. Damit sind flexible Kontrollflüsse mit beinahe beliebigen Transitionen zwischen den einzelnen Zuständen möglich. Es können Strukturen entstehen, die durch wohlgeformte Schachtelung von Blöcken, wie sie aus strukturierten Programmiersprachen bekannt sind, nicht mehr abgebildet werden können. Verzweigungen etwa können zu einer beliebigen anderen Aktivität führen, was unter Umständen ohne eine Graphstruktur nur schwer auszudrücken wäre. Darum muss die Prozessbeschreibungssprache ermöglichen, den Kontrollfluss als Graphstruktur abzuspeichern.

3. Aufruf vorhandener Services bzw. Komponenten als Aktionen: Genau wie in den Aktivitäten des Prozessdiagramms muss es auch in der XML-Beschreibung eine Möglichkeit geben, auf Softwarekomponenten des statischen Modells zu verweisen, so dass bei der Ausführung eine Bindung vorgenommen und eine konkrete Instanz aufgerufen werden kann.

4. Übergabe von Parametern an Komponenten: Beim Aufruf einer Komponente müssen Parameter übergeben werden können. Für dieses Konstrukt der Prozessdiagramme ist eine adäquate Repräsentation in der XML-Beschreibung nötig.

5. Beschreibung von Nebenläufigkeit und Synchronisation: Eine parallele Ausführung mehrerer Kontrollflussstränge und ihre Synchronisation muss analog zu den Prozessdiagrammen auch in der XML-Beschreibung spezifizierbar sein.

6. Beschreibung von Entscheidungsbedingungen an Verzweigungen: Eine Transition nach einer Verzweigung kann im Prozessdiagramm eine Bedingung bzw. Guard enthalten. Entsprechend müssen Transitionen in der XML-Beschreibung mit Entscheidungsbedingungen auszustatten sein.

7. Schachtelung von Prozessteilen: Analog zur Hierarchisierung von Prozessdiagrammen muss ein ähnlicher Mechanismus für die XML-Beschreibung der Workflows vorhanden sein: Andere Prozesse, die ihrerseits als XML-Dokument beschrieben sind, müssen zum einen aufgerufen sowie zum anderen mit ihrer Beschreibung direkt in den übergeordneten Prozess eingebettet werden können.

8. Angabe von Ports zur Modellierung von Dokumenttypen: Entsprechend dem Port-Konzept für die Modellierung der Dokumenttypen in Prozessdiagrammen (vgl. Abschnitt 6.3) müssen Portdefinitionen in die Prozessbeschreibung integriert werden. Diese Informationen werden sowohl bei der Modellierung als auch bei der Ausführung benötigt, um die Konsistenz von Dokumenttypen zu gewährleisten.

9. Typisierung von Transitionen: Wie in Abschnitt 6.3.2 vorgestellt, werden die Transitionen in Prozessdiagrammen typisiert. Demzufolge müssen in der XML-Beschreibung der Workflows die möglichen Transitionstypen sowohl entsprechend ihrer Definition in Kapitel 6 und 7 festgelegt als auch den einzelnen Transitionen zugewiesen werden können.

10. Repräsentation aller sonstigen Sprachelemente von Prozessdiagrammen: Da in der XML-Beschreibung Prozessdiagramme kodiert werden sollen, ohne dass Informationen verloren gehen, müssen außer den bis jetzt genannten Punkten auch alle weiteren Sprachelemente, die in Kapitel 7 definiert wurden, in dem XML-Dokument gespeichert werden. Hierunter fallen zum Beispiel Informationen über die Transaktionseigenschaften eines Prozesses.

11. Format möglichst als bekannter Standard: Es wäre von Vorteil, wenn die Prozessbeschreibungssprache nicht proprietär entwickelt würde, sondern auf einem allgemein anerkannten Standard beruhte. Dadurch würde sowohl die Unterstützung durch andere Werkzeuge als auch die Austauschbarkeit mit anderen Partnern erleichtert.

12. Konkrete Dokumente für Benutzer möglichst verständlich und lesbar: Zwecks besserer Wartbarkeit sollten die zu dem Beschreibungsformat gehörigen XML-Dokumente für einen Benutzer lesbar und verständlich sein, so dass er prinzipiell leichte Änderung auch direkt an der textlichen Beschreibung vornehmen kann, ohne das Workflow-Modell im Entwurfswerkzeug visuell zu bearbeiten.

13. Keine Definition von eigentlich nicht benötigten Elementen: Eine Vielzahl von Elementdefinitionen, die zur Repräsentation von Prozessdiagrammen eigentlich nicht benötigt werden, erhöhen die Komplexität, erschweren die Verständlichkeit und sollten deshalb vermieden werden. Diese Anforderung kommt besonders bei der Betrachtung vorhandener Sprachen und Standards zum Tragen, die unter Umständen über die Konstrukte von Prozessdiagrammen weit hinausgehen.

8.2 Vorhandene XML-Formate für Prozessmodelle

Es gibt zahlreiche XML-Formate, in denen sich Workflow-Modelle textuell abspeichern lassen; einige von ihnen sind sogar als Industriestandard etabliert. Da es gemäß einer der oben aufgestellten Anforderungen geboten ist, solche Standards zu unterstützen, wollen wir in diesem Abschnitt stellvertretend zwei Formate vorstellen und gegen die Anforderungen evaluieren. Es handelt sich dabei um *XLANG*, einem Format, in dem beim BizTalk Server 2000 die Prozessdefinitionen abgelegt werden, und dem *XML Metadata Interchange (XMI)* Konzept zur Speicherung objekt-orientierter Modelle.

8.2.1 Beschreibung von Prozessen mit XLANG

An dieser Stelle soll die von Microsoft entwickelte Prozessbeschreibungssprache *XLANG* vorgestellt werden. Wie bereits im Abschnitt 4.1 erwähnt wurde, wird sie innerhalb des Microsoft BizTalk Systems verwendet, um die modellierten Workflows in einer maschinell interpretierbaren Form zu speichern. Laut [Tha01] handelt es sich bei XLANG um eine XML-basierte Sprache zur Spezifikation des Nachrichtenaustausches zwischen interagierenden Webservices. Sie ist vor allem als Basis für die automatisierte Ausführung von unternehmensübergreifenden Geschäftsprozessen gedacht. Dabei liegt das Augenmerk auf der Modellierung des nach außen hin sichtbaren Verhaltens durch den Austausch von Nachrichten, nicht jedoch die internen Verarbeitungsschritte der beteiligten Komponenten. XLANG stellt eine Erweiterung der *Web Service Description Language* (WSDL, siehe [WSDL01]) dar. WSDL wurde vom W3-Konsortium entwickelt, um die externen Schnittstellen von Netzwerkdiensten, d.h. insbesondere Webservices spezifizieren zu können. Während durch eine WSDL-Beschreibung lediglich angegeben wird, welche Operationen ein Webservice anbietet bzw. welche Nachrichtentypen er verarbeitet kann, ergänzt XLANG eine solche Spezifikation um die Beschreibung der Abfolge verschiedener Nachrichten. Auf diese Weise wird ein Protokoll festgelegt, das die komplette Interaktion mit einem Webservice bestimmt. Zwar geht es bei den in dieser Arbeit betrachteten XML-Prozessen nicht um die Integration von Webservices, die verwendeten Softwarekomponenten könnten jedoch prinzipiell als Webservices interpretiert werden. Bei der folgenden Beschreibung der wichtigsten Sprachkonstrukte von XLANG werden insbesondere die Anforderungen aus Abschnitt 8.1 berücksichtigt.

Statisches Modell

Bevor ein Workflow-Ablauf mit XLANG beschrieben werden kann, müssen die zu integrierenden Softwarekomponenten in einem statischen Modell spezifiziert werden. Dies geschieht mit Hilfe von WSDL, wobei für jede Komponente sogenannte *Ports* definiert werden. Jeder Port ist dabei eine Menge von Nachrichtentypen, die von der Komponente versendet oder empfangen werden können. Die Spezifikation der Nachrichtentypen geschieht mit Hilfe von XML-Schemata. Dieser Ansatz ist für die Umsetzung des Port-Konzepts in XML-Prozessen nicht optimal geeignet, da es Komponenten geben kann, die keine genau festgelegte Dokumentstruktur fordern, sondern lediglich bestimmte enthaltene Elemente (vgl. Abschnitt 6.3.1). Das von uns eingeführte Port-Konzept ist somit flexibler und kann nicht auf WSDL abgebildet werden.

Beschreibung der Prozessabläufe

Aufbauend auf dem statischen Modell kann nun der Ablauf eines Workflows festgelegt werden. Grundbaustein bei der XLANG-Beschreibung eines Prozesses ist die sogenannte Aktion. Dabei handelt es sich um das Ansprechen eines bestimmten Ports und der Auswahl einer seiner Operationen. Eine Operation ist dabei das Senden oder Empfangen einer Nachricht. Alternativ stehen auch Aktionen zur Verfügung, mit denen sich zeitliches Verhalten modellieren lässt, wie beispielsweise das Verzögern des Ablaufs um ein bestimmtes Zeitintervall. Im Abschnitt 8.1 wurde gefordert, dass Aktionen parametrisierbar sein müssen. Zwar ist es in XLANG möglich, zu einer Operation eine Menge von Parametern zu definieren, als Werte können jedoch nur Inhalte der Nachrichten übergeben werden, die bei Ausführung der Operation ausgetauscht werden. Die Aktionen einer Prozessbeschreibung können durch verschiedene XML-Elemente zu beliebigen Abläufen aggregiert werden. XLANG kennt zu diesem Zweck Kontrollflusskonstrukte, wie sie aus Programmiersprachen bekannt sind. Tabelle 8.1 zeigt die wichtigsten XLANG-Elemente. Für eine detaillierte Beschreibung aller Sprachelemente sei auf [Tha01] verwiesen.

XML-Element	Kontrollfluss
<xlang:sequence>	Sequenz
<xlang:switch>	bedingte Verzweigung
<xlang:all>	Parallele Ausführung
<xlang:while>	Iteration
<xlang:exception>	Fehler- bzw. Ausnahmebehandlung

Tab. 8.1: Sprachelemente von XLANG

Es ist in XLANG nicht möglich, Ablaufbeschreibungen graphorientiert anzugeben, d.h. als Menge von Aktionen mit Transitionen untereinander. Ebenfalls kann für den Übergang von einer Aktion zur nächsten keine Typisierung der Transition angegeben werden. Dies wäre jedoch zur Umsetzung des Konzepts der typisierten Transitionen unbedingt notwendig. Um eine Schachtelung von XLANG-Beschreibungen zu erreichen, können lediglich XLANG-Elemente in ein anderes XLANG-Dokument eingebettet, nicht jedoch auf ein anderes XLANG-Dokument verwiesen werden. Abbildung 8.1 zeigt als Beispiel einen Ausschnitt des Prozesses aus Abschnitt 6.2, wie er als XLANG-Beschreibung aussehen könnte.

```
<switch>
    <branch>
        <case>/transaktion/type='ueberweisung'</case>
        <sequence>
            <all>
                <sequence>
                    <switch>
                        <branch>
                            <case>
                                /transaktion/auftraggeber/name
                                =/transaktion/empfaenger/name and not(
                                /transaktion/empfaenger/bankleitzahl
                                ='47262703'
                            </case>
```

```
                        <sequence>
                               <action operation="SQLQuery" .../>
                               <action operation="SendMail" .../>
                        </sequence>
                    </branch>
                </switch>
            </sequence>
            <sequence>
                ...
            </sequence>
        </all>
        <action operation="XSLTTransformer" .../>
    </sequence>
  </branch>
  ...
</switch>
```

Abb. 8.1: XLANG-Beschreibung eines Beispielprozesses

Es gibt eine Reihe weiterer XML-Sprachen für die maschinelle Ausführung von Workflows, wie beispielsweise die *Business Process Modeling Language* (BPML, siehe [Ark01]), die von der Business Process Management Initiative (BPMI) entwickelt wurde. Die BPMI ist eine Organisation mehrerer IT-Unternehmen, die sich zum Ziel gesetzt hat, Workflow-Management-Technologien zu standardisieren. BPML und anderen Sprachen liegen ähnliche Ansätze wie XLANG zu Grunde. Aus diesem Grund soll an dieser Stelle auf eine detaillierte Betrachtung dieser Prozessbeschreibungssprachen verzichtet werden.

8.2.2 Der XML Metadata Interchange Standard (XMI)

Der von der OMG spezifizierte Standard *XML Metadata Interchange* (XMI, siehe [XMI00]) stellt einen Lösungsversuch für die Austauschproblematik objektorientierter Modellierungssprachen dar. Über die Festlegung auf einen gemeinsamen Industriestandard für die textuelle Darstellung von objektorientierten Modellen hinaus handelt es sich um einen generativen Ansatz, mit dem sich für verschiedene Metamodelle automatisch XML-Formate erzeugen lassen. Daher soll an dieser Stelle untersucht werden, inwiefern mit diesem Standard ein Format für das Speichern von Prozessdiagrammen gewonnen werden könnte.

Bei der Entwicklung von XMI wurde ursprünglich ein einheitliches Textformat gesucht, mit dem sich in UML erstellte Modelle zwischen verschiedenen Modellierungswerkzeugen und CASE-Tools austauschen ließen. Bei dem Entwurf eines entsprechenden XML-Formats kam man schnell zu der Notwendigkeit eines generativen Ansatzes, um der raschen Weiterentwicklung von UML Rechnung zu tragen und auch zukünftige Erweiterungen flexibel in das Austauschformat integrieren zu können (vgl. [Jec00]). So ist ein Konzept entstanden, mit dem sich für alle derzeitigen und zukünftigen Sprachen, die auf der Meta Object Facility (MOF, siehe [MOF00]) der OMG basieren, ein Austauschformat für Metadaten erzeugen lässt. MOF ist dabei „das gemeinsame Meta-Metamodell aller objektorientierten Modelle der OMG Sprachhierarchie“ (siehe [Jec00]). So ist zum Beispiel das UML-Metamodell eine Ausprägung des MOF-Modells (vgl. die Vier-Schichten-Architektur der UML, Abschnitt 7.1.1).

Konkret definiert XMI mit den sogenannten *DTD Production Rules* Prinzipien

zur Abbildung von Klassendiagrammen in DTD-Strukturen. Ausprägungen des Klassendiagramms können so in einem zur DTD gültigen XML-Dokument gespeichert werden, welches man auch *XMI-Dokument* nennt. Wie solche Modell-Ausprägungen in das XMI-Dokument überführt werden können, wird ebenfalls in XMI durch sogenannte *Document Production Rules* festgelegt (siehe [XMI00], 1-1). Da sowohl das MOF-Modell als auch das UML-Metamodell als Klassendiagramm dargestellt werden, konnte die Nutzbarkeit von XMI durch die Erzeugung je einer DTD für UML und MOF demonstriert werden. Wegen ihrer großen Bedeutung wurden diese zwei DTDs als Anhang in die XMI-Spezifikation aufgenommen. Eine Ausprägung des UML-Metamodells, also jedes UML-Modell bestehend aus Klassen- und sonstigen Diagrammen, kann somit nun textuell in einem XMI-Dokument gespeichert werden, das bezüglich der UML-DTD gültig ist (vgl. Abbildung 8.2).

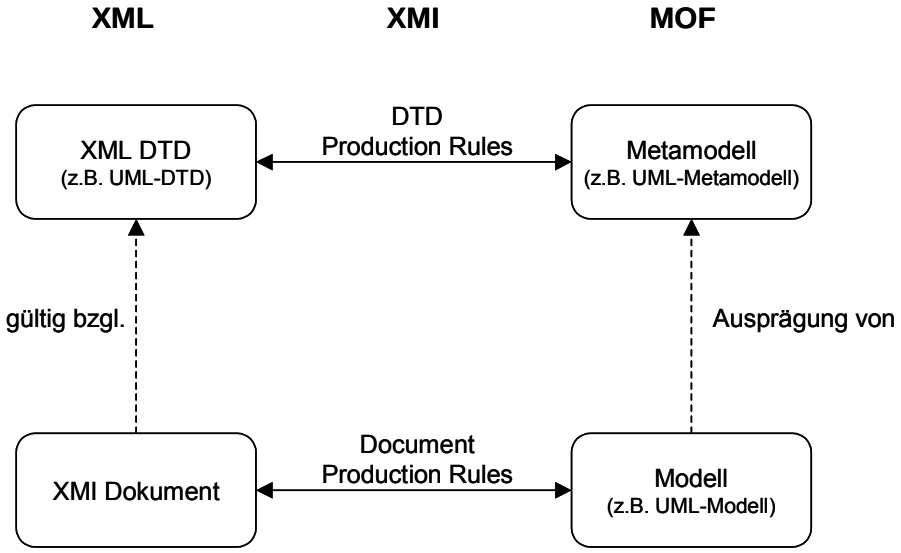


Abb. 8.2: Zusammenhang zwischen XMI, XML, MOF und UML

Die Produktionsregeln zur Abbildung eines Metamodells in eine DTD enthalten Generierungsschablonen, mit denen sich die Grundprimitive des MOF wie Klassen, Attribute und Assoziationen in XML-Strukturen überführen lassen (vgl. [Jec00]). Dabei wird zum Beispiel für jede Klasse und für jedes Attribut der Klasse ein eigenes XML-Element unter Beibehaltung der eindeutigen Namensgebung definiert. Die Elemente für die Attribute sind dem Element für die Klasse untergeordnet und werden von diesem umschlossen. Als Inhaltsmodell werden soweit möglich die Attributdatentypen des Klassendiagramms beibehalten, skalare Datentypen werden als PCDATA definiert. Aufzählungsdatentypen werden durch ein leeres Element mit einem Attribut abgebildet, bei dem der DTD-Aufzählungsmechanismus benutzt wird. Über den jeweiligen Rollennamen lassen sich auch Assoziationen zu anderen Klassen als Unterelement eines Klassenelements definieren, wobei im Inhaltsmodell auf ein Element für die assoziierte Klasse verwiesen wird. Dadurch können im XMI-Dokument die Ausprägungen der Klassen entsprechend ihrer Beziehungen hierarchisch geschachtelt werden. Vererbungsbeziehungen werden im Wesentlichen durch einfaches Kopieren der Elemente aus der Elternklasse realisiert.

Derzeit laufen Initiativen zur Portierung der XMI-Prinzipien von DTDs auf den Nachfolgestandard XML-Schema. Es ist damit zu rechnen, dass in einer zukünftigen

Version von XMI die bei XML-Schemata vorgesehene Verfeinerung der Datentypen und die explizite Definition von Vererbungsmechanismen zu einer verbesserten Abbildung der Metamodelle auf XML-Formate führen wird.

Für die Speicherung von Prozessdiagrammen in einem XML-Format bieten sich mit XMI zwei mögliche Ansätze: Als erweitertes UML-Diagramm könnte man ein Prozessdiagramm als XMI-Dokument speichern, das den Spezifikationen der standardisierten UML-DTD (siehe [XMI00] A-1) genügt, oder man generiert aus dem Metamodell für Prozessdiagramme eine eigene DTD und erzeugt dazu gültige XMI-Dokumente. Zunächst soll auf Letzteres eingegangen werden.

Das Metamodell für Prozessdiagramme (siehe Abschnitt 7.3) ist im Wesentlichen eine Teilmenge des UML-Metamodells, ergänzt um einige Stereotypen, die man als virtuelle Metaklassen auffassen kann. Virtuell bedeutet in diesem Fall, dass sie bei der Modellierung zwar zur Klassifizierung von Elementen in der Modellebene benutzt werden können, quasi so, als seien sie selbst Elemente des Metamodells. In Wirklichkeit befinden sich die Stereotyp-Definitionen als Ausprägungen der Metaklasse Stereotype jedoch nicht in der Metamodell-Ebene sondern ebenfalls in der Modellebene. Es handelt sich bei dem Profil für Prozessdiagramme als Erweiterung gemäß den vorgegebenen Erweiterungsmechanismen also nur um eine scheinbare Ergänzung des UML-Metamodells. Weil somit Prozessdiagramme durch das UML-Metamodell und das spezielle Profil für Prozessdiagramme auf zwei verschiedenen Ebenen definiert sind (siehe Abbildung 8.3), ist es nicht möglich, mit Hilfe von XMI eine DTD zu erzeugen, die alle Metamodell-Elemente für Prozessdiagramme inklusive der Stereotypen repräsentiert.

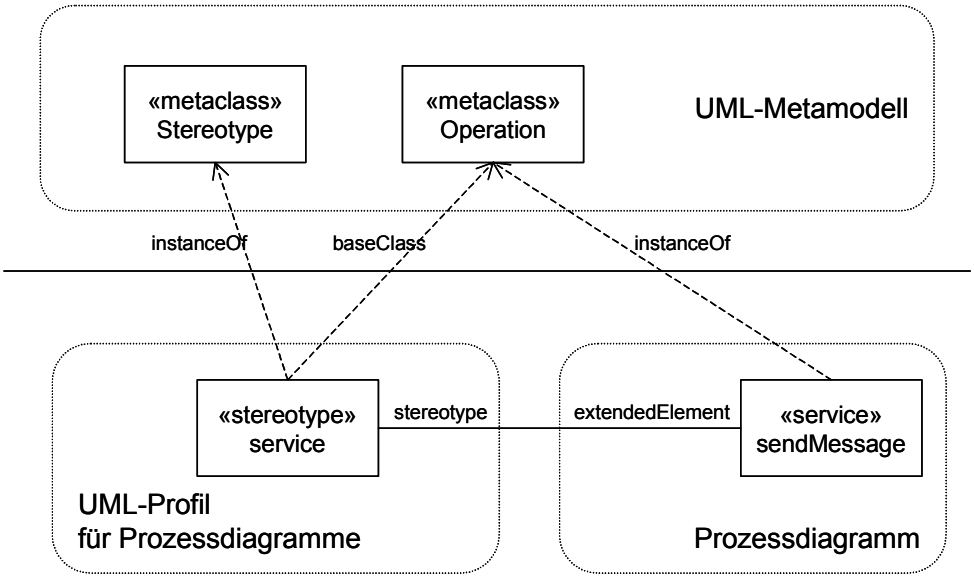


Abb. 8.3: Zusammenhang von UML-Profil und -Metamodell zur Definition von Prozessdiagrammen

Da die Möglichkeit einer eigenen Prozessdiagramm-DTD nach der XMI-Spezifikation nicht gegeben ist, bleibt alternativ noch die Betrachtung der vorgegebenen UML-DTD. Nach dieser DTD lassen sich beliebige Ausprägungen des UML-Metamodells in einem XMI-Dokument abspeichern. Die folgende Abbildung zeigt exemplarisch ein gekürztes Dokument für ein UML-Modell mit einem einfachen Aktivitätendiagramm (zum besse-

ren Verständig der Elementnamen vgl. Abbildung 7.1 zum Metamodell für Aktivitätendiagramme). Zwecks besserer Lesbarkeit wurde insbesondere darauf verzichtet, die Elemente entsprechend der DTD vollständig mit ihren UML-Package-Präfixen zu bezeichnen. Durch die Präfixe wird zwar eine eindeutige Benennung sichergestellt, doch werden die XML-Marken auch sehr lang und für den Leser schwerer verständlich.

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>

  <XMI.content>

       <ActivityGraph xmi.id="xmi.3">
           <name>workflowProcessingActivityGraph</name>

           <StateMachine.top>
              <CompositeState xmi.id="xmi.4">
                  <name>activities_top</name>
                  <CompositeState.subvertex>

                    <Pseudostate xmi.id="xmi.5">
                      <name>initalState</name>
                      <Pseudostate.kind xmi.value="initial"/>
                      <StateVertex.outgoing>
                        <Transition xmi.idref="xmi.6"/>
                      </StateVertex.outgoing>
                    </Pseudostate>

                    <ActionState xmi.id="xmi.7">
                      <name>exampleAction</name>
                      <StateVertex.outgoing>
                        <Transition xmi.idref="xmi.8"/>
                      </StateVertex.outgoing>
                      <StateVertex.incoming>
                        <Transition xmi.idref="xmi.6"/>
                      </StateVertex.incoming>
                    </ActionState>

                    <FinalState xmi.id="xmi.11">...</FinalState>

                  </CompositeState.subvertex>
              </CompositeState>
           </StateMachine.top>

           <StateMachine.transitions>
               <Transition xmi.id="xmi.6">
                  <Transition.stateMachine>
                    <StateMachine xmi.idref="xmi.3"/>
                  </Transition.stateMachine>
                  <Transition.source>
                    <StateVertex xmi.idref="xmi.5"/>
                  </Transition.source>
                  <Transition.target>
                    <StateVertex xmi.idref="xmi.7"/>
                  </Transition.target>
              </Transition>
              <Transition xmi.id="xmi.8">...</Transition>
           </StateMachine.transitions>

       </ActivityGraph>

  </XMI.content>
</XMI>
```

Abb. 8.4: Ein kleines UML-Aktivitätendiagramm als XMI-Dokument

Wie erwartet, korrespondiert das XMI-Dokument sehr stark mit der Struktur des UML-Metamodells. Da ein Aktivitätendiagramm dort als Vernetzung von Zuständen durch Transitionen definiert ist, wird die Anforderung einer graphorientierten Strukturierung des XML-Formats (vgl. Abschnitt 8.1 Punkt 2) erfüllt: Das Beispiel zeigt, wie die Ablaufbeschreibung im Gegensatz zu einem durch XLANG beschriebenen Workflow nicht durch Strukturelemente wie `<sequence>`, `<while>` oder `<all>`, sondern durch wechselseitige Referenzierungen auf nachfolgende und vorausgehende Elemente innerhalb des Ablaufgraphen definiert ist. Außerdem ist es mit der UML-DTD inhärent möglich, alle Konstrukte von Prozessdiagrammen abzuspeichern, die keine Erweiterung von Aktivitätendiagrammen darstellen. Dazu gehören der Aufruf von Operationen eines Interfaces mit Übergabe von Parametern in Aktivitäten, die Schachtelung von Abläufen durch Subaktivitäten sowie Konstrukte für Nebenläufigkeit, Synchronisation, Verzweigung und die Angabe von Ausdrücken als Entscheidungsbedingungen (guards).

Da die Definition der Erweiterungen im Zusammenhang mit dem UML-Profil für Prozessdiagramme konform mit dem UML-Metamodell vorgenommen wurde, ist es möglich, auch die Erweiterungen und die neuen Attribute abzuspeichern, die sich durch die Stereotypen und Tag-Definitionen (siehe Kapitel 7) ergeben. Da sich die Stereotyp-Definitionen jedoch wie aus Abbildung 8.3 ersichtlich auf der gleichen Modellhierarchie-Ebene wie die Prozessdiagramme selbst befinden, werden sie ebenfalls im XMI-Dokument als Instanzen der Metaklasse Stereotype abgelegt. Die Dokumentverknüpfungs- und Referenzierungsmechanismen XLink und XPointer erlauben es, ein getrenntes XMI-Dokument für diese Definitionen des Erweiterungsprofils anzulegen und mit den XMI-Dokumenten für die konkreten Prozessdiagramme zu verknüpfen (siehe [XMI00] Kapitel 3.8). Auf diese Art ist es möglich, die vorgenommenen Spracherweiterungen zu integrieren und alle Sprachelemente von Prozessdiagrammen einschließlich ergänzender Attribute, etwa für die Angabe von Ports, Transitionstypen oder Transaktionen, in XMI-Dokumenten abzulegen.

Die UML-DTD liefert eine Repräsentation des gesamten UML-Metamodells und ist daher sehr umfangreich. Entsprechend können auch die XMI-Dokumente sehr viele Details bezüglich eines UML-Modells enthalten, was im Zusammenhang mit den schon erwähnten langen Markenbezeichnungen zu schlechter Lesbarkeit der Dokumente für einen menschlichen Leser führt. Insofern wird die Anforderung, möglichst keine unnötigen Elemente zu definieren, nicht erfüllt, aber dadurch angenähert, dass die meisten Elemente in der DTD als optional deklariert wurden und auch ausgelassen werden können, wenn sie zum Beispiel für Prozessdiagramme nicht relevant sind.

8.2.3 Bewertung

Nachdem exemplarisch die zwei Sprachansätze vorgestellt wurden, soll nun eine zusammenfassende Bewertung erfolgen. Dabei wird überprüft, inwieweit sie den im Abschnitt 8.1 aufgestellten Anforderungen gerecht werden. In Tabelle 8.2 sind alle Anforderungen und die Bewertung der einzelnen Ansätze in Kurzform aufgelistet.

Anforderung	XLANG	XMI
XML-Format	+	+
Graphorientierte Beschreibung des Kontrollflusses	-	+
Aufruf vorhandener Services bzw. Komponenten	+	+
Übergabe von Parametern an Komponenten	o	+
Beschreibung von Nebenläufigkeit und Synchronisation	+	+
Entscheidungsbedingungen an Verzweigungen	+	+
Schachtelung von Prozessteilen	o	+
Angabe von XML-Dokumenttypen mit Ports	o	+
Typisierung von Transitionen	-	+
Repräsentation sonstiger Sprachelemente	-	+
Bekanntes Format oder Standard	+	+
Dokumente verständlich und lesbar	+	-
Keine Definition nicht benötigter Elemente	-	-

+ = vorhanden - = nicht vorhanden o = teilweise vorhanden

Tab. 8.2: Bewertung vorhandener XML-Formate für Prozessmodelle

XLANG kommt als XML-basierte Sprache zur Beschreibung von Prozessen grundsätzlich in Frage. Sie ist zudem für einen menschlichen Leser recht gut verständlich und kann aufgrund ihrer Verwendung in Microsoft BizTalk als bekannte Sprache angesehen werden. XLANG ist jedoch sehr stark auf die Beschreibung von Webservices und insbesondere die Interaktion per Nachrichtenaustausch fokussiert. Beide Aspekte sind im Zusammenhang mit XML-Prozessen nicht unmittelbar von Bedeutung. Dadurch ergäbe sich bei der Repräsentation von Prozessdiagrammen in XLANG ein Overhead, da die XLANG-Beschreibung viele nicht erforderliche Elemente enthalten würde. Weiterhin erfolgt die Spezifizierung der Workflow-Abläufe nicht graphbasiert, dies wurde jedoch im Abschnitt 8.1 ausdrücklich gefordert. Insgesamt gesehen ist XLANG daher für die Beschreibung von XML-Prozessen nicht optimal geeignet.

Dagegen erfüllt die UML-DTD der XMI-Spezifikation alle Muss-Bestimmungen des Anforderungskataloges. Weil jedoch für die Definition von Prozessdiagrammen ein ganz kleiner Teilbereich des gesamten UML-Metamodells ausreicht und die Benutzung der langen und daher unleserlichen Elementnamen sowie das Abspeichern vieler UML-Details vermieden werden soll, werden wir trotz der sonst positiv bewerteten Eigenschaften von XMI im folgenden Abschnitt mit PML ein eigenes, angepasstes XML-Format definieren. Da XMI und die darin enthaltene UML-DTD aber ein allgemein anerkannter, zunehmend wichtiger Industriestandard ist und inzwischen von vielen CASE-Tools als Datenformat für Metadaten unterstützt wird, wäre eine Konvertierungsmöglichkeit zwischen XMI- und PML-Dokumenten sinnvoll. Dazu könnte zum Beispiel zusätzlich zur Definition von PML ein XSL-Stylesheet zur Konvertierung von und nach XMI angegeben werden, womit es unter der Voraussetzung, dass ein CASE-Tool die Spezifikationen von UML und XMI voll unterstützt, möglich wäre, auch Prozessdiagramme mit diesem Werkzeug zu editieren. Da mit den Erweiterungsmechanismen und Stereotypen jedoch ein Randbereich der UML betreten wird, fehlt bei vielen Werkzeugen noch deren vollständige Unterstützung. Diese Tatsache hat die Entwicklung eines eigenen Editors für Prozessdiagramme nötig gemacht (vgl. Kapitel 9).

8.3 Process Markup Language (PML)

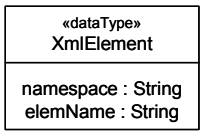
In diesem Abschnitt soll nun die von uns entwickelte XML-basierte Prozessbeschreibungssprache PML (Process Markup Language) vorgestellt werden, die speziell zur textuellen Repräsentation von Prozessdiagrammen vorgesehen ist. Dazu definieren wir PML als XML-Sprache durch Angabe eines XML-Schemas (siehe [XSch01]). Alle Informationen eines Prozessdiagramms sind dabei in der zugehörigen PML-Beschreibung enthalten. Somit ist es möglich, aus einem PML-Dokument ein Prozessdiagramm zu rekonstruieren. Soweit möglich wurden die Einschränkungen, die für Prozessdiagramme in Form von UML-Constraints formuliert wurden, mit Hilfe von Sprachkonstrukte wie Vererbung, Fallunterscheidungen und Kardinalitäten, die für XML-Schemata zur Verfügung stehen, umgesetzt. Dadurch kann von vornherein ausgeschlossen werden, dass eine PML-Beschreibung bestimmte Fehler im Modell enthält, oder zumindest würden diese bereits von einem XML-Parser beim Einlesen eines PML-Dokuments erkannt.

Die folgenden Beschreibungen gehen jeweils von den aus Abschnitt 7.3 bekannten UML-Stereotypen aus und geben die zugehörige Umsetzung nach PML in Form eines XML-Schema-Fragments an. Das vollständige PML-Schema ist auf der CD-ROM zu dieser Diplomarbeit zu finden. Damit die Erläuterungen möglichst anschaulich und leicht nachvollziehbar werden, ist zu jedem Stereotypen ein beispielhafter PML-Abschnitt angegeben. Zunächst soll betrachtet werden, wie die von uns eingeführten Ports in PML beschrieben werden. Es handelt sich dabei um ein wichtiges Konzept von Prozessdiagrammen, das in mehreren Stereotypen zum Einsatz kommt. Anschließend wird die PML-Beschreibung des statischen Modells und schließlich von Prozessen selbst vorgestellt.

8.3.1 Beschreibung von Ports

Datentyp XmlElement

Der Grundbaustein für die Beschreibung eines Ports sind XML-Elemente, die durch den UML-Datentyp XmlElement repräsentiert werden sollen. Dieser besitzt die beiden Attribute „elemName“ und „namespace“, die den Namen und den Namensraum des Elements angeben. Daraus leitet sich folgender Typ im XML-Schema für PML ab:

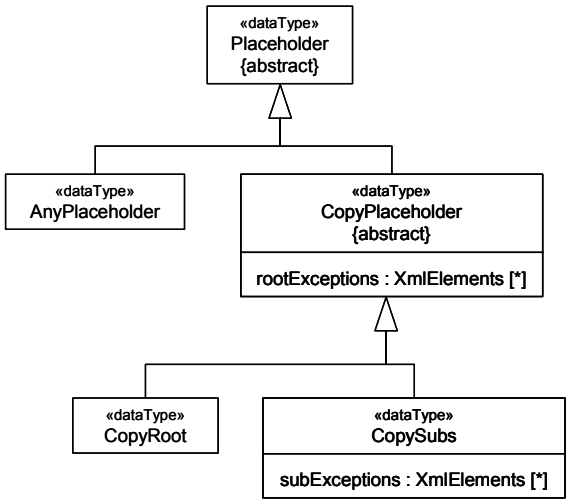



```
<xsd:complexType name="ElementType">
      <xsd:attribute name="element" type="xsd:string"
                     use="required"/>
      <xsd:attribute name="namespace" type="xsd:string"/>
</xsd:complexType>
```

Abb. 8.5: PML-Schema-Fragment für XmlElement

Platzhalter

Statt konkrete XML-Elemente zu spezifizieren, können in Ports auch sogenannte Platzhalter vorkommen. Es gibt drei Arten, zu denen die in Abbildung 8.6 dargestellten XML-Typen Any, CopyRoot und CopySubs gehören. Für CopyRoot und CopySubs kann jeweils durch <except> eine Menge von XML-Elementen definiert werden, die beim Binden nicht übernommen werden sollen. Es ist zu beachten, dass zur Beschreibung eines CopySubs-Platzhalters in PML zwei XML-Elemente der Typen CopyRootType und CopySubsType kombiniert werden müssen, damit sowohl das Attribut rootExceptions als auch subExceptions abgebildet werden kann (vgl. Abbildung 8.8).




```
<xsd:complexType name="AnyType"/>

<xsd:complexType name="CopyRootType">
      <xsd:sequence>
            <xsd:element name="except" type="ElementType"
                         minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="CopySubsType">
      <xsd:sequence>
            <xsd:element name="except" type="ElementType"
                         minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>
```

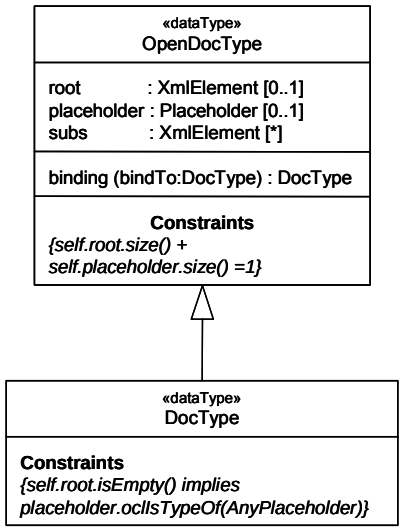
Abb. 8.6: PML-Schema-Fragment für Platzhalter

Datentypen DocType und OpenDocType

Entsprechend des Datentyps OpenDocType hat jeder Dokumenttyp genau ein Wurzelelement, das in PML im Element <root> beschrieben wird, und kann ein oder mehrere Unterelemente fordern, die durch <sub> beschrieben werden. Statt des Elements <root>, das ein konkret benanntes Wurzelelement definiert, kann auch der Platzhalter <any> eingesetzt werden. Es kann jedoch nicht ein Wurzelelement und ein Platzhalter gleichzeitig angegeben werden. Diese Bedingung spiegelt sich in der Semantik des

XML-Schema-Elements `<xsd:choice>` wider.

Nur falls es sich um einen OpenDocType handelt, lässt sich neben <any> auch der Platzhalter <copyRoot> verwenden. Zur Erfüllung dieser Anforderung wurde der spezielle XML-Typ OpenDoctypeType definiert. Durch die Verwendung des optionalen <copySubs>-Elements wird beschrieben, dass neben dem Wurzelelement auch die Unterelemente übernommen werden sollen.




```
<xsd:complexType name="DoctypeType">
      <xsd:sequence>
            <xsd:choice>
                  <xsd:element name="root" type="ElementType"/>
                  <xsd:element name="any" type="AnyType"/>
            </xsd:choice>
            <xsd:element name="sub" type="ElementType"
                        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="OpenDoctypeType">
      <xsd:sequence>
            <xsd:choice>
                  <xsd:element name="root" type="ElementType"/>
                  <xsd:element name="any" type="AnyType"/>
                  <xsd:sequence>
                        <xsd:element name="copyRoot"
                                    type="CopyRootType"/>
                        <xsd:element name="copySubs"
                                    type="CopySubsType"
                                    minOccurs="0"/>
                  </xsd:sequence>
            </xsd:choice>
            <xsd:element name="sub" type="ElementType"
                        minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>
```

Abb. 8.7: PML-Schema-Fragment für Dokumenttypen

Abbildung 8.8 zeigt zur Verdeutlichung beispielhaft die Beschreibung zweier Dokumenttypen:

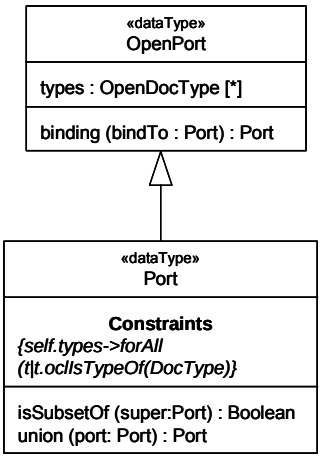
```
<doctype>
  <root element="root1" namespace="..."/>
  <sub element="child1" namespace="..."/>
  <sub element="child2" namespace="..."/>
</doctype>

<doctype>
  <copyRoot>
    <except element="root2" namespace="..."/>
    ...
  </copyRoot>
  <copySubs>...</copySubs>
</doctype>
```

Abb. 8.8: Beispiele für die PML-Beschreibung von Dokumenttypen

Datentypen Port und OpenPort

Ein Port schließlich ist eine Menge von DocTypes, ein OpenPort eine Menge von OpenDocTypes. Diese Definition ergibt direkt die Umsetzung für das PML-Schema:




```
<xsd:complexType name="OpenPortType">
      <xsd:sequence>
            <xsd:element name="doctype" type="OpenDoctypeType"
                         maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="PortType">
      <xsd:sequence>
            <xsd:element name="doctype" type="DoctypeType"
                         maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>
```

Abb. 8.9: PML-Schema-Fragment für Ports

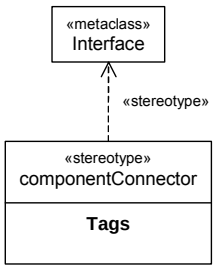
8.3.2 Beschreibung des statischen Modells

Sowohl die Bestandteile des statischen Modells von XML-Prozessen, d.h. Komponenten und Transitionstypen, als auch die Prozesse selbst, sind in Anlehnung an UML in *Packages* gegliedert (vgl. Abschnitt 7.4). Dies führt zu einer wesentlich besseren Beherrschbarkeit umfangreicher Modelle. Da die Beschreibung von komplexen Prozessen recht groß werden kann, wurde PML so konstruiert, dass nicht alle Bestandteile eines Package in ein und demselben Dokument beschrieben werden. Vielmehr werden alle statischen Elemente eines Package in einem gemeinsamen PML-Dokument definiert, jeder Prozess besitzt jedoch ein eigenes Dokument. Daraus ergeben sich zwei globale XML-Elemente, die als Wurzelelement eines PML-Dokuments benutzt werden können: <package> zur Beschreibung des statischen Modells und <process>, um einen Prozess zu beschreiben.

Um die statischen Elemente eines Package zu definieren, erhält ein PML-Dokument also als Wurzelelement das Tag <package>. Als Attribut „name“ wird der Name des Package angegeben. Unterhalb davon können sich zwei Elemente <components> und <transitionTypes> befinden, um die Komponenten bzw. die Konnektor-Klassen und die Transitionstypen zu beschreiben.

Stereotyp «componentConnector»

Einer Komponente, die in Prozessdiagrammen verwendet werden kann, ist im UML-Modell stets ein Stereotyp «componentConnector» als erweitertes Interface zugeordnet. Für jeden solchen Konnektor befindet sich unterhalb des Elements <components> ein Element mit dem Namen <component>. Als Wert seines Attributs „interface“ gibt man den Namen des zugehörigen UML-Interface an, allerdings ohne das Package, in dem es enthalten ist. Dieses ergibt sich bereits aus dem Package-Kontext, in dem der Konnektor beschrieben wird. Daneben enthält ein Konnektor eine Menge von Services als Methoden des Interface (siehe nächster Absatz). Der Aufbau des XML-Elements ist im Schema folgendermaßen definiert:

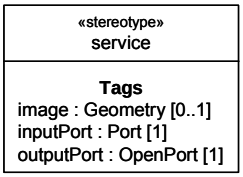



```
<xsd:complexType name="ComponentType">
      <xsd:sequence>
            <xsd:element name="service" type="ServiceType"
                         minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="interface" type="xsd:string"
                     use="required"/>
</xsd:complexType>
```

Abb. 8.10: PML-Schema-Fragment für Konnektoren

Stereotyp «service»

Um die Services einer Komponente zu definieren, dient das Element <service> innerhalb der Beschreibung eines Konnektors. Sein Attribut „method“ spezifiziert den Namen der zugehörigen Methode im Interface der Komponente. Jeder Service hat nun vier Bestandteile, die sich aus den Werten des UML-Stereotyps «service» ergeben: Im Attribut „href“ des Elements <image> kann eine URL bestimmt werden, die auf eine Grafik zeigt. Diese wird im Modellierungswerkzeug zur Darstellung des Service verwendet. Die Elemente <inputPort> und <outputPort> legen die Ein- und Ausgangsports des Service fest. Ihr Aufbau entspricht der Definition aus Abschnitt 8.3.1. Schließlich kann ein Service beliebig viele Parameter besitzen. Ihre Namen, im UML-Modell in der Metaklasse Parameter festgelegt, werden durch das Attribut „name“ der Elemente <parameter> bestimmt. Die Angabe eines Typs ist nicht erforderlich, da implizit der Typ „String“ vorausgesetzt wird (vgl. Abschnitt 6.2.2). Es ist jedoch darauf zu achten, dass die Reihenfolge der <parameter>-Elemente mit der Reihenfolge der Parameter in der zugehörigen Methodensignatur übereinstimmen muss.




```
<xsd:complexType name="ServiceType">
      <xsd:sequence>
            <xsd:element name="image" type="ImageType"
                         minOccurs="0"/>
            <xsd:element name="inputPort" type="PortType"/>
            <xsd:element name="outputPort" type="OpenPortType"/>
            <xsd:element name="parameter" type="ParameterType"
                         minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="method" type="xsd:string"
                     use="required"/>
</xsd:complexType>
```

Abb. 8.11: PML-Schema-Fragment für Services

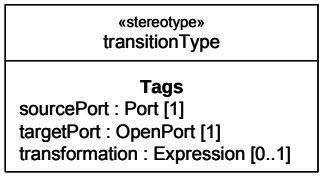
Abbildung 8.12 zeigt beispielhaft die Beschreibung eines Konnektors:

```
<package name="de.sundn.sunflowexamples">
  <components>
    <component interface="IComponent1">
      <service method="serviceA">
        <image href="..."/>
        <inputPort>...</inputPort>
        <outputPort>...</outputPort>
        <parameter name="par1"/>
      </service>
    </component>
  </components>
</package>
```

Abb. 8.12: Beispiel für die PML-Beschreibung eines Konnektors

Stereotyp «transitionType»

Für jeden Transitionstyp des Package existiert ein Unterelement <transitionType> unterhalb von <transitionTypes>. Durch sein Attribut „name“ legt man den Namen des Typs fest, wobei das Package nicht angegeben wird. Dieses ergibt sich bereits aus dem Package-Kontext, in dem der Transitionstyp beschrieben wird. Aus dem Stereotyp «transitionType» ergibt sich, dass ein Transitionstyp aus drei Teilen besteht: Durch die Elemente <sourcePort> und <targetPort> lassen sich Ein- und Ausgangsports definieren. Dies geschieht analog zu Abschnitt 8.3.1. Im Anschluss daran gibt das Element <transformation> das XSL-Stylesheet an, mit dem das Migrationsdokument transformiert werden soll, wenn es über eine Transition des beschriebenen Typs läuft. Dazu gibt es zwei Möglichkeiten: Entweder verweist das Attribut „href“ auf eine URL, unter der das gewünschte Stylesheet abgelegt ist, oder das Stylesheet wird direkt als Unterelement von <transformation> in der PML-Beschreibung notiert.




```
<xsd:complexType name="TransitionTypeType">
      <xsd:sequence>
            <xsd:element name="sourcePort" type="PortType"/>
            <xsd:element name="targetPort" type="OpenPortType"/>
            <xsd:element name="transformation"
                         type="TransformationType"
                         minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name"
                     type="xsd:string"
                     use="required"/>
</xsd:complexType>
```

Abb. 8.13: PML-Schema-Fragment für Transitionstypen

Abbildung 8.14 zeigt beispielhaft die Beschreibung eines Transitionstyps:

```
<package name="de.sundn.sunflowexamples">
  ...
  <transitionTypes>
    <transitionType name="Type1">
      <sourcePort>...</sourcePort>
      <targetPort>...</targetPort>
      <transformation href="..."/>
    </transitionType>
    <transitionType name="Type2">
      <sourcePort>...</sourcePort>
      <targetPort>...</targetPort>
      <transformation
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:stylesheet>...</xsl:stylesheet>
      </transformation>
    </transitionType>
    ...
  </transitionTypes>
</package>
```

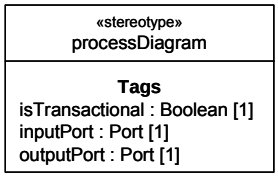
Abb. 8.14: Beispiel für die PML-Beschreibung eines Transitionstyps

8.3.3 Beschreibung von Prozessen

Stereotyp «processDiagram»

Wenn mit einem PML-Dokument ein Workflow beschrieben werden soll, hat es das Wurzelelement <process>. Entsprechend dem zugehörigen Stereotyp «processDiagram» gibt das Attribut „name“ den Namen einschließlich des Package an, zu dem das Diagramm gehört, und „transactional“, ob der Prozess als Transaktion auszuführen ist oder nicht. Erlaubte Werte sind hier „true“ und „false“. Ein- und Ausgangsport des Prozesses werden in den Unterelementen <inputPort> und <outputPort> entsprechend Abschnitt 8.3.1 definiert. Darüber hinaus enthält das Element <process> Unterelemente <state>, von denen jedes einen Zustand des Prozessdiagramms mit dem Stereotypen «stateWithPorts» beschreibt.

Bei Bedarf kann außerdem ein Element <extensions> angegeben werden. Sein Inhalt ist im PML-Schema durch das Konstrukt <xsd:any> als *offener Inhalt* (siehe [XSch01] 5.5) spezifiziert, d.h. es können hier beliebige XML-Daten eingebettet werden. Hierdurch wird PML zu einer erweiterbaren Sprache, die die Möglichkeit bietet, PML-Dokumente um zusätzliche Informationen zu ergänzen. Dies können beispielsweise Daten sein, die von einem bestimmten Werkzeug benötigt werden. So nutzt das im Kapitel 9 beschriebene Modellierungswerkzeug den Bereich <extensions>, um Koordinaten für alle Zustände zu speichern, damit der Benutzer stets die gleiche grafische Darstellung der Workflow-Modelle vorfindet. Es ergibt sich folgende XML-Schema-Definition für die PML-Beschreibung eines Prozessdiagramms:




```
<xsd:complexType name="ProcessType">
      <xsd:sequence>
            <xsd:element name="inputPort" type="PortType"/>
            <xsd:element name="outputPort" type="OpenPortType"/>
            <xsd:element name="state" type="StateType"
                         minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="extensions" type="ExtensionsType"
                         minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:string"
                     use="required"/>
      <xsd:attribute name="transactional" type="xsd:boolean"
                     use="required"/>
</xsd:complexType>

<xsd:complexType name="ExtensionsType">
      <xsd:sequence>
            <xsd:any minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
</xsd:complexType>
```

Abb. 8.15: PML-Schema-Fragment für Prozessdiagramme

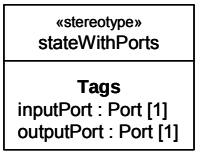
Stereotyp «stateWithPorts»

Jedes Element <state> beschreibt einen Zustand, dem das Stereotyp «stateWithPorts» zugeordnet ist. Gemäß diesem Stereotypen existieren zwei Unterelemente <inputPort> und <outputPort>, um die Ports des Zustands anzugeben. Als drittes Unterelement steht eines der weiter unten beschriebenen PML-Elemente, die jeweils den Zustandstyp widerspiegeln, der sich aus der UML-Metaklasse des Zustands ergibt. Schließlich haben wir aus technischen Gründen ein Attribut „id“ eingeführt. Dieses ist vom XML-Datentyp ID und muss eine innerhalb des Prozesses eindeutige Identifizierung des Zustands erlauben. Es hat jedoch keine Entsprechung im UML-Modell des Prozesses.

Die Verkettung der einzelnen Zustände entsprechend der Transitionen geschieht mit Hilfe eines Elements <next>, das jeweils auf einen Nachfolgezustand verweist. Es steht innerhalb der typabhängigen PML-Elemente und wird im Zusammenhang mit Transitionen weiter unten beschrieben.

Das Stereotyp «stateWithPorts» besitzt eine Reihe von Constraints, die Aussagen über die Ports in Abhängigkeit vom Typ des Zustands machen. Diese Einschränkungen lassen sich nicht in einem XML-Schema ausdrücken, da es sich dabei um eine Grammatik handelt, die lediglich die Struktur sowie die Wertebereiche der Inhalte von XML-Dokumenten spezifiziert. Es existiert jedoch kein Mechanismus, mit dem sich Bedingungen an die konkreten Inhalte eines Dokuments formulieren lassen. Eine PML-Dokument kann daher im Prinzip unzulässige Zustandsbeschreibungen enthalten.

Aus Platzgründen sind in der folgenden Abbildung mehrere XML-Typen nicht dargestellt, sie können auf der CD-ROM zur Arbeit nachgesehen werden:




```
<xsd:complexType name="StateType">
      <xsd:sequence>
            <xsd:element name="inputPort" type="PortType"/>
            <xsd:element name="outputPort" type="PortType"/>
            <xsd:choice>
                  <xsd:element name="initial"
                               type="InitialType"/>
                  <xsd:element name="final"
                               type="FinalType"/>
                  <xsd:element name="fork"
                               type="ForkType"/>
                  <xsd:element name="join"
                               type="JoinType"/>
                  <xsd:element name="decision"
                               type="DecisionType"/>
                  <xsd:element name="merge"
                               type="MergeType"/>
                  <xsd:element name="serviceCall"
                               type="ServiceCallType"/>
                  <xsd:element name="subactivity"
                               type="SubactivityType"/>
                  <xsd:element name="synchfork"
                               type="SynchforkType"/>
                  <xsd:element name="synchjoin"
                               type="SynchjoinType"/>
                  <xsd:element name="synch"
                               type="SynchType"/>
                  </xsd:choice>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:ID"
                           use="required"/>
      </xsd:complexType>
      <xsd:complexType name="ServiceCallType">
            <xsd:sequence>
                  <xsd:element name="callservice"
                               type="CallserviceType"/>
                  <xsd:element name="next" type="NextType"/>
            </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="SubactivityType">
            <xsd:sequence>
                  <xsd:element name="process"
                               type="SubProcessType"/>
                  <xsd:element name="next" type="NextType"/>
            </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="CallserviceType">
            <xsd:sequence>
                  <xsd:element name="argument"
                               type="ArgumentType"
                               minOccurs="0"
                               maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="component"
                           type="xsd:string"
```

```
                        use="required"/>
        <xsd:attribute name="method" type="xsd:string"
                        use="required"/>
</xsd:complexType>
<xsd:complexType name="ArgumentType">
        <xsd:simpleContent>
                <xsd:extension base="xsd:string">
                        <xsd:attribute name="parameter"
                                        type="xsd:string"
                                        use="required"/>
                </xsd:extension>
        </xsd:simpleContent>
</xsd:complexType>
```

Abb. 8.16: PML-Schema-Fragment für Zustände

Initial: Durch das Element <initial> wird der Startzustand eines Prozesses bestimmt. Es besitzt ein Unterelement <next>, mit dem eine Transition ohne Guard zum Nachfolgezustand definiert wird.

Final: Das Element <final> bestimmt den Endzustand des Prozesses. Es erlaubt keine Unterelemente.

Fork: Mit Hilfe des Elements <fork> werden parallele Teilprozesse beschrieben. Durch <next> können beliebig viele Nachfolgezustände definiert werden, mindestens jedoch zwei. Dabei können Guards angegeben werden, um bedingte Parallelität auszudrücken.

Join: Mit dem Element <join> lassen sich parallele Teilprozesse wieder vereinigen. Es muss eine Transition zu einem Nachfolgezustand angegeben werden, die keinen Guard enthalten darf.

Decision: Unter einem solchen Zustand mit dem Element <decision> verstehen wir einen UML-Zustand vom Typ „junction“, mit dem der Prozessablauf in einen von mehreren alternativen Strängen verzweigt. Daher sind, wie schon bei Fork, beliebig viele Nachfolgezustände erlaubt, mindestens jedoch zwei. Es müssen Guards angegeben werden, wobei darauf zu achten ist, dass zur Laufzeit stets eine der Guard-Bedingungen wahr wird oder eine Transition den vordefinierten Guard „else“.

Merge: Dies ist ein Zustand vom Typ „junction“ in dem verschiedene Prozessstränge zusammenlaufen. Das Element <merge> muss eine Transition zu einem Nachfolgezustand enthalten, die keinen Guard besitzen darf.

Subaktivität: Mit Hilfe des Elements <subactivity> lässt sich ein Subprozess aufrufen. Dieser wird durch das Unterelement <process> ausgewählt. Es kann entweder ein Attribut „ref“ enthalten, dessen Wert einen anderen Prozess einschließlich seines Package referenziert, oder es beschreibt einen vollständigen Prozess. Dieser ist jedoch als „lokaler“ Prozess anzusehen, der nicht an anderer Stelle explizit angesprochen werden kann. Ein Subaktivitätszustand besitzt eine Transition zu einem Nachfolger ohne Angabe eines Guards.

```
<subactivity>
  <process ref="de.sundn.sunflowexamples.Process2"/>
  <next>...</next>
</subactivity>

<subactivity>
  <process name="Subprocess1" transactional="false">
    <inputPort>...</inputPort>
    <outputPort>...</outputPort>
    <state id="id1">...</state>
    ...
  </process>
  <next>...</next>
</subactivity>
```

Abb. 8.17: Zwei Möglichkeiten zum Ansprechen eines Subprozesses

Service: Durch Verwendung des Elements <service> lässt sich ein Zustand mit dem UML-Stereotypen «serviceState» beschreiben. Neben der Transition zu einem Nachfolgezustand ohne Guard, enthält es ein Unterelement <callservice>, mit dem ein Service aufgerufen wird. In seinen Attributen „component" und „method" wird der Name eines Konnektors einschließlich des Package sowie einer seiner Services ausgewählt. Diese Bestandteile müssen selbstverständlich eine Entsprechung in der statischen PML-Beschreibung des Package haben. Zu jedem Parameter des ausgewählten Service muss <callservice> ein Unterelement <argument> enthalten. In seinem Attribut „parameter" gibt man an, zu welchem Parameter es gehört. Als Textinhalt des Elements muss ein Ausdruck angegeben werden, der vom Prozessinterpreter ausgewertet werden kann (vgl. dazu Abschnitt 10.2.4).

```
<service>
  <callservice component="de.sundn.sunflowexamples.IComponent1"
               method="serviceA">
    <argument parameter="par1">...</argument>
    ...
  </callservice>
  <next>...</next>
</service>
```

Abb. 8.18: Aufruf eines Service

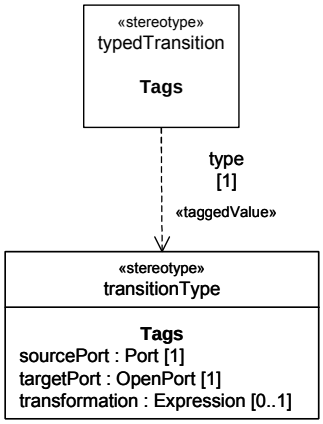
Synch: Ein Synchronisationszustand wird durch das Element <synch> definiert. Er enthält mittels <next> eine Transition zum zugehörigen Synchjoin-Zustand (siehe unten), die keinen Guard enthalten darf.

Synchfork: Unter einem Synchfork-Zustand verstehen wir einen UML-Fork-Zustand, dessen einer Nachfolger ein Synchronisationszustand ist. Wir treffen diese Unterscheidung aus technischen Gründen, damit der Zustand sich auf einfache Weise von einem gewöhnlichen Fork-Zustand unterscheiden lässt. Durch die Unterscheidung bereits auf Modellebene werden Fallunterscheidungen beispielsweise zur Ausführungszeit des Prozessinterpreters vermieden. Das Element <synchfork> hat Transitionen zu genau zwei Nachfolgern, wobei es sich um einen Synchronisationszustand und einen anderen Zustand handeln muss. Dabei sind keine Guards erlaubt.

Synchjoin: Dabei handelt es sich um einen Join-Zustand, dessen einer Vorgänger ein Synchronisationszustand ist. Wie schon beim Synchfork bringt diese Unterscheidung auf Modellebene Vorteile mit sich. Das Element <synchjoin> hat eine Transition zu einem Nachfolger, wobei keine Guards erlaubt sind.

Stereotyp «typedTransition»

Wie bereits angedeutet dient zur Verkettung der verschiedenen Zustände eines Prozessdiagramms das Element <next>. Eine solche Verkettung entspricht einer UML-Transition mit dem Stereotyp «typedTransition». Dabei ist zu unterscheiden zwischen Transitionen mit und ohne Guard. Zunächst besitzt das Element <next> ein Attribut „ref“ vom XML-Datentyp IDREF. Es lässt sich damit auf ein Element <state> als Zielzustand der Transition verweisen, das ja als eindeutige Identifikation ein Attribut „id“ besitzt. Außerdem muss mit Hilfe des Attributs „transitionType“ ein existierender Transitionstyp mit Package ausgewählt werden, der den Typ des Zustandsübergangs festlegt. Wenn ein Guard erlaubt ist, so hat <next> zusätzlich ein Unterelement <guard>, dessen Textinhalt ein Ausdruck ist, der vom Prozessinterpreter ausgewertet werden kann (vgl. Abschnitt 10.2.2). Dazu wird mit Hilfe der Vererbung mittels <xsd:extension> der XML-Typ GuardedNextType definiert. Die beiden Constraints des Stereotyps, die Aussagen über die Gültigkeit der Ports machen, können, wie bereits weiter oben argumentiert, nicht im XML-Schema umgesetzt werden, da sie sich auf Inhalte, und nicht die Struktur eines PML-Dokuments beziehen.




```
<xsd:complexType name="NextType">
      <xsd:attribute name="ref" type="xsd:IDREF"
                     use="required"/>
      <xsd:attribute name="transitionType" type="xsd:string/>
</xsd:complexType>
<xsd:complexType name="GuardedNextType">
      <xsd:complexContent>
            <xsd:extension base="NextType">
                  <xsd:sequence>
                        <xsd:element name="guard"
                                     type="xsd:string"/>
                  </xsd:sequence>
            </xsd:extension>
      </xsd:complexContent>
</xsd:complexType>
```

Abb. 8.19: PML-Schema-Fragment für Transitionen

Zum Abschluss zeigt Abbildung 8.20 die verkürzte PML-Beschreibung des aus Abschnitt 8.2.2 bereits bekannten, einfachen Prozessdiagramms:

```
<process name="de.sundn.sunflowexamples.Short"
         transactional="false">
  <inputPort>...</inputPort>
  <outputPort>...</outputPort>
  <state id="id1">
    <inputPort>...</inputPort>
    <outputPort>...</outputPort>
    <initial>
      <next ref="id2" transitionType="..."/>
    </initial>
  </state>
  <state id="id2">
    <inputPort>
      <doctype>
        <root element="..."
              namespace="..."/>
      </doctype>
    </inputPort>
    <outputPort>...</outputPort>
    <service>
      <callservice component="..."
                   methode="exampleAction"/>
      <next ref="id3" transitionType="..."/>
    </service>
  </state>
  <state id="id3">
    <inputPort>...</inputPort>
    <outputPort>...</outputPort>
    <final/>
  </state>
</process>
```

Abb. 8.20: Ein kleines Prozessdiagramm als PML-Dokument

8.3.4 Bewertung

Nachdem die einzelnen Sprachbestandteile von PML in den letzten beiden Abschnitten vorgestellt wurden, soll nun eine Bewertung der Sprache erfolgen. Da PML speziell für die textuelle Repräsentation von Prozessdiagrammen entwickelt wurde, sind alle Muss-Bestimmungen erfüllt. Es lassen sich alle Sprachelemente von Prozessdiagrammen in PML ausdrücken und rekonstruieren, um aus einer PML-Beschreibung wieder ein visuelles Modell zu generieren. Die einzige Forderung, die nicht erfüllt werden konnte, ist die nach einer bekannten oder sogar standardisierten Sprache. Falls Workflow-Modelle mit Hilfe einer solchen bekannten Sprache wie beispielsweise XMI beschrieben werden sollen, muss ggf. die Möglichkeit einer Konvertierung zwischen PML und dieser Sprache geschaffen werden. Dazu könnte beispielsweise ein XSL-Stylesheet entwickelt werden.

Aufgrund der Eigenschaften von XML-Schemata ließen sich nicht alle Einschränkungen bzw. Bedingungen, die für Prozessdiagramme als OCL-Constraints formuliert wurden, in der Definition der Sprache PML umsetzen. Daher ist es prinzipiell möglich, mit PML Prozesse zu beschreiben, die keine gültigen Prozessdiagramme darstellen. Nachdem also ein PML-Dokument maschinell eingelesen wurde, sollte vor der weiteren Verarbeitung mit Hilfe eines geeigneten Validierers (siehe Abschnitt 7.4.2) überprüft werden, ob es ein korrektes Modell enthält. Tabelle 8.3 fasst noch einmal alle Aspekte im Bezug auf eine Prozessbeschreibungssprache zusammen.

Anforderung	**PML**
XML-Format	+
Graphorientierte Beschreibung des Kontrollflusses	+
Aufruf vorhandener Services bzw. Komponenten	+
Übergabe von Parametern an Komponenten	+
Beschreibung von Nebenläufigkeit und Synchronisation	+
Entscheidungsbedingungen an Verzweigungen	+
Schachtelung von Prozessteilen	+
Angabe von XML-Dokumenttypen mit Ports	+
Typisierung von Transitionen	+
Repräsentation sonstiger Sprachelemente	+
Bekanntes Format oder Standard	-
Dokumente verständlich und lesbar	+
Keine Definition nicht benötigter Elemente	+

+ = vorhanden - = nicht vorhanden

Tab. 8.3: Bewertung von PML bezüglich der Anforderungen

8.4 Werkzeuge für die PML-Verarbeitung

Im vorhergehenden Abschnitt wurde die Abbildung zwischen dem in 7.3 eingeführten Metamodell und der Sprache PML definiert. Um beide Formate in realen Systemen nutzen zu können, benötigt man Werkzeuge, die die Formate entsprechend den Abbildungsvorschriften ineinander überführen können. Dazu werden im Folgenden die von uns entwickelten Werkzeuge für jede der beiden Transformationsrichtungen vorgestellt.

8.4.1 PML-Importer

Zunächst soll der *PML-Importer* beschrieben werden, der in der Lage ist, eine PML-Beschreibung einzulesen und daraus eine Objektstruktur entsprechend den Datenklassen aus Abschnitt 7.4 aufzubauen. Mit Hilfe des Importers können Prozesse oder Packages importiert werden, wobei sich die resultierenden Objekte sowohl in ein bereits im Speicher befindliches, zu einem früheren Zeitpunkt importiertes Prozessmodell einfügen, als auch zum Aufbau eines neuen Prozessmodells benutzen lassen.

Da die internen Abläufe des Importers sehr technisch sind, soll im Folgenden nur grob seine Arbeitsweise erläutert werden. Abbildung 8.21 zeigt die Klasse *PMLImporter*:

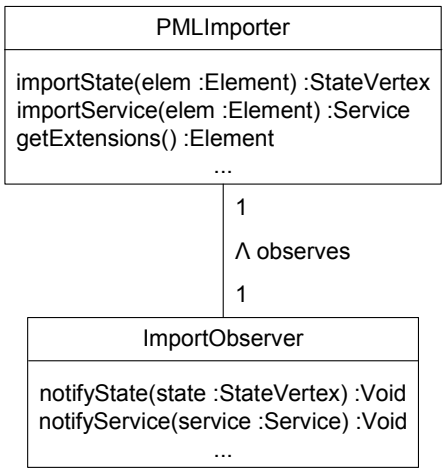


Abb. 8.21: Klassen PMLImporter und ImportObserver

Die Klasse PMLImporter besitzt zu jeder Datenklasse aus Abschnitt 7.4 eine Methode, die als Parameter das XML-Element erhält, in dem das korrespondierende Modellelement beschrieben wird, und als Rückgabewert ein Objekt der zugehörigen Datenklasse liefert. Beispielsweise konstruiert die Methode `importService(..)` aus der PML-Beschreibung eines Service eine Objektinstanz der Klasse Service.

Der Importer durchläuft Schritt für Schritt das PML-Dokument, das importiert werden soll, und ruft dabei die passenden Methoden auf. Auf diese Weise wird sukzessive die gesamte Objektstruktur aufgebaut. Eine Besonderheit ergibt sich beim Import von Transitionen zwischen Prozesszuständen. Da es passieren kann, dass der Importer im PML-Dokument an eine Stelle kommt, die eine Transition beschreibt, deren Zielzustand bis zu diesem Zeitpunkt noch gar nicht importiert wurde, kann die zugehörige

Objektstruktur noch nicht vollständig aufgebaut werden. Während des Importvorgangs merkt sich der Importer daher zunächst zu jeder Transition die eindeutige Knotenidentifikation (siehe Abschnitt 8.3.3) ihres Zielzustands. Erst wenn das gesamte PML-Dokument verarbeitet ist, also alle Zustände importiert wurden, wird zu jeder Transition der Zielzustand gesetzt.

Je nachdem zu welchem Zweck ein Prozessmodell importiert werden soll, müssen während des Importvorgangs zusätzliche Berechnungen stattfinden. So muss beispielsweise der Prozesseditor zu jedem Modellelement die von ihm benötigten Adapter erzeugen (vgl. Abschnitt 7.4, Flexible Zugriffsschicht auf Elemente des Datenmodells) und im Falle eines Prozesszustands zusätzlich die Koordinaten für die grafische Darstellung des Diagramms laden. Um den Importer für solche Anforderungen flexibel zu gestalten, haben wir unter Einsatz des Entwurfsmusters *Observer*, das dazu dient, Nachrichtenempfänger über bestimmte Ereignisse zu informieren (siehe [Gam96] S.287), die Klasse *ImportObserver* realisiert, zu der der Importer eine Assoziation besitzt. Immer wenn ein Modellelement importiert wurde, wird die zugehörige Methode des Observers aufgerufen und bekommt das Modellelement übergeben. Man muss eine Unterklasse von ImportObserver implementieren, die die Methoden überschreiben und die gewünschten Berechungen durchführt. Falls dazu Daten aus dem Extensions-Bereich des PML-Dokuments (siehe Abschnitt 8.3.3) benötigt werden - wie es bei den Koordinaten der Fall ist - kann der Observer diese mit Hilfe der Methode `getExtensions()` des Interpreter abfragen.

8.4.2 PML-Exporter

Der PML-Exporter dient dazu, aus einem Prozessmodell, das sich in Form einer Objektstruktur im Speicher befindet, PML-Beschreibungen zu generieren, die dauerhaft gespeichert werden können. Es lassen sich sowohl einzelne Prozesse und Packages als auch ein gesamtes Prozessmodell exportieren. Die internen Abläufe des Exporters sind sehr technisch, so dass im Folgenden nur grob die Arbeitsweise erläutert wird. Abbildung 8.22 zeigt die Klasse *PMLExporter*:

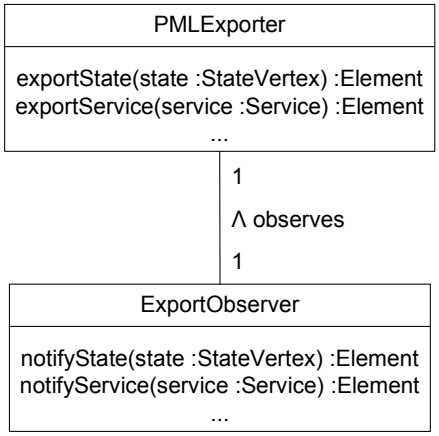


Abb. 8.22: Klassen PMLExporter und ExportObserver

Die Klasse PMLExporter besitzt zu jeder Datenklasse aus Abschnitt 7.4 eine Methode, die als Parameter eine Instanz der jeweiligen Datenklasse erhält und als Rückgabewert

ein XML-Element liefert, das eine PML-Beschreibung des Modellelements darstellt. Beispielsweise konstruiert die Methode `exportService(..)` aus einem Service-Objekt die zugehörige PML-Beschreibung. Der Exporter durchläuft in einer Tiefensuche die zu exportierenden Modellteile und ruft dabei die passenden Methoden auf. Auf diese Weise wird schrittweise das PML-Dokument generiert.

Im Abschnitt 8.3.3 wurde die Möglichkeit beschrieben, mit Hilfe des Extension-Abschnitts in einem PML-Dokument zusätzliche Informationen abzulegen. Der Prozesseditor speichert dort beispielsweise die Koordinaten aller Prozesszustände in der grafischen Darstellung. Analog zum Importer kommt auch hier das Entwurfsmuster *Observer* zum Einsatz (siehe [Gam96] S.287]). Die Klasse *ExportObserver*, zu der der Exporter eine Assoziation besitzt, wird immer dann benachrichtigt, wenn ein Modellelement exportiert wurde. Dies geschieht durch Aufruf der zugehörigen Methode des Observers, die das Modellelement übergeben bekommt. Als Rückgabewert kann die Methode ein XML-Element liefern, das der Exporter als Kind des Elements `<extensions>` in das PML-Dokument einbettet. Man muss bei Bedarf eine Unterklasse von ExportObserver implementieren, die die Methoden überschreibt und die gewünschten XML-Elemente konstruiert.

Unter Verwendung der hier vorgestellten Prozessbeschreibungssprache PML und den beiden Konvertierungswerkzeugen zwischen PML und der aus Abschnitt 7.4 bekannten Datenstruktur folgt im nächsten Kapitel die Vorstellung des Prozesseditors, mit dem sich Prozessdiagramme entwerfen lassen.

9 Modellierungswerkzeug für Prozessdiagramme

Nachdem in den vorausgegangenen Kapiteln das Konzept der Prozessdiagramme, sowie nützliche Werkzeuge und Verfahren beschrieben wurden, stellt dieses Kapitel ein Modellierungwerkzeug vor, mit dem sich Workflow-Modelle und Prozessdiagramme entwerfen lassen. Es wurde von uns in der Sprache Java entwickelt. Im ersten Abschnitt folgt zunächst eine allgemeine Erläuterung des Zwecks eines Prozesseditors sowie der zu Grunde gelegten Konzepte. Im Anschluss daran wird auf den Aufbau der Benutzungsoberfläche und grundlegende Bedienungstechniken eingegangen, bevor in einer Art Tutorial beschrieben wird, wie man schrittweise einen Workflow modelliert. Der letzte Abschnitt des Kapitels betrachtet die Implementierung des Systems.

9.1 Der Prozesseditor

Die Aufgabe des Prozesseditor besteht darin, dem Modellierer ein Werkzeug zur Verfügung zu stellen, mit dem sich komplette Workflow-Modelle verwalten lassen. Dazu zählt zunächst die Möglichkeit, ein statisches Modell anzulegen, d.h. Konnektoren zu erzeugen und Transitionstypen zu definieren, die als Grundbausteine für Prozessdiagramme benötigt werden. Weiterhin gibt es die Möglichkeit, aus diesen Bestandteilen komplexe Prozesse zu konstruieren bzw. vorhandene Prozesse zu bearbeiten. Zu diesem Zweck stellt der Editor Prozessdiagramme visuell dar und erlaubt direkte Änderungen an dieser Darstellung durch den Modellierer. Insbesondere stehen Eingabedialoge zur Verfügung, über die sich die aus Abschnitt 6.3 bekannten Ports spezifizieren lassen. Um dem Modellierer diese Arbeit soweit wie möglich abzunehmen, verwendet der Editor intern das Verfahren der automatischen Portberechnung (siehe Abschnitt 7.6). Lediglich im statischen Modell müssen die Ports von Hand eingegeben werden, bei der Konstruktion von Prozessdiagrammen findet jedoch wann immer möglich eine sukzessive Berechnung der Ports statt.

Weiterhin kann der Modellierer vom Prozesseditor aus jederzeit den Validierungsalgorithmus aus Abschnitt 7.5 starten, um zu überprüfen, ob sein Diagramm Fehler enthält bzw. an welchen Stellen die Modellierung noch unvollständig ist. Der Editor selbst verwendet den Algorithmus intern, um sicherzustellen, dass keine ungültigen Prozessdiagramme dauerhaft gespeichert werden. Gegebenenfalls wird der Modellierer durch Fehlermeldungen auf die Inkonsistenzen aufmerksam gemacht.

Damit der Prozesseditor in der Lage ist, eine solche Verwaltung von Prozessen zu erlauben, muss er intern auf einer geeigneten Datenstruktur arbeiten, mit der sich Prozessdiagramme repräsentieren lassen. Ausgehend von dem in Kapitel 8 erläuterten XML-Format PML zur Speicherung von Prozessdiagrammen wäre ein erster Ansatz für ein internes Datenmodell, direkt auf den XML-Dokumenten und -Elementen zu arbeiten, die nach dem Parsen eines PML-Dokuments beispielsweise als Ausprägung des *Document Object Models* (DOM, siehe [DOM01]) im Speicher vorliegen könnten und zugreifbar wären. Der Vorteil dieser Lösung läge in der einfachen Integration vorliegender PML-Prozessbeschreibungen: Man müsste sie lediglich parsen und könnte dann

direkt auf den Daten arbeiten. Analog wäre es sehr einfach, PML-Dokumente zu erzeugen: Man bräuchte nur die interne Struktur zu serialisieren und als XML-Dokument abzuspeichern. Trotzdem haben wir uns gegen diese Lösung entschieden, weil die hierarchische Struktur der PML-Dokumente nicht für eine effiziente Verarbeitung innerhalb der Prozessmodellierung geeignet ist. Man müsste ständige Suchanfragen an und Traversierungen über den PML-Baum durchführen, was die Implementierung erschwert und zudem höhere Laufzeiten erzwingt. Außerdem sind vorhandene Implementierungen für das DOM bislang noch sehr speicherineffizient.

Aus diesen Gründen wird stattdessen die im Abschnitt 7.4 vorgeschlagene Implementierung des erweiterten UML-Metamodells für Prozessdiagramme verwendet. Sie ist für die Zwecke des Editor effizienter als die direkte XML-Verarbeitung, da alle Beziehungen zwischen den Elementen eines Prozesses direkt durch Assoziationen zwischen den Datenklassen realisiert sind. Da somit für die interne Datenstruktur keine PML-Dokumente zu Grunde gelegt werden, ergibt sich jedoch die Notwendigkeit für Konvertierungswerkzeuge zwischen PML-Dokumenten und dem internen Datenmodell. Für diese Aufgaben wurden im Abschnitt 8.4 die beiden Werkzeuge PML-Importer und PML-Exporter vorgestellt, die vom Editor zum Speichern und Laden von Prozessmodellen verwendet werden. Die Implementierung des Editors umfasst ca. 15.000 LOC (lines of code).

9.2 Bedienung des Prozesseditors

Wenn der Prozesseditor gestartet wird, erscheint das Hauptfenster. Es enthält eine Menüleiste mit dem Eintrag „File“. Dort lässt sich das Programm beenden oder eine der beiden Aktionen „New Project“ oder „Load Project“ auswählen. Durch „New Project“ kann man ein neues Projekt (siehe Abschnitt 9.2.1) anlegen. Es erscheint ein Eingabedialog, in dem der gewünschte Name eingegeben werden muss. Dieses Projekt ist dann geöffnet und kann bearbeitet werden. Alternativ lässt sich mit „Load Project“ ein bestehendes Projekt öffnen. Mit Hilfe eines Dateiauswahldialogs kann die gewünschte Projektdatei ausgewählt werden. Nach beiden Aktionen präsentiert sich die Oberfläche des Programms wie in Abbildung 9.1 gezeigt.

Neben der Menüleiste, die nach wie vor die beiden oben genannten Aktionen anbietet, besteht das Hauptfenster aus zwei Teilen. Auf der linken Seite befindet sich der sogenannte Projektbaum (siehe Abschnitt 9.2.2), der alle Bestandteile des geöffneten Projekts zeigt. Im rechten Bereich des Fensters erfolgen die Benutzereingaben. Hier werden abhängig davon, welcher Bestandteil des Projekts im Projektbaum angewählt ist, Eingabemasken eingeblendet, die Veränderungen am Workflow-Modell ermöglichen (siehe Abschnitt 9.2.3). Dieser Grundaufbau des Editors bleibt während der gesamten Bearbeitung bestehen. Beim Aufruf einiger Aktionen erscheinen Dialoge, in denen zunächst die jeweiligen Eingaben getätigt werden müssen, bevor die restliche Benutzungsoberfläche wieder aktiviert wird (siehe Abschnitt 9.2.4). Solche Dialoge werden üblicherweise als *modale Dialoge* bezeichnet.

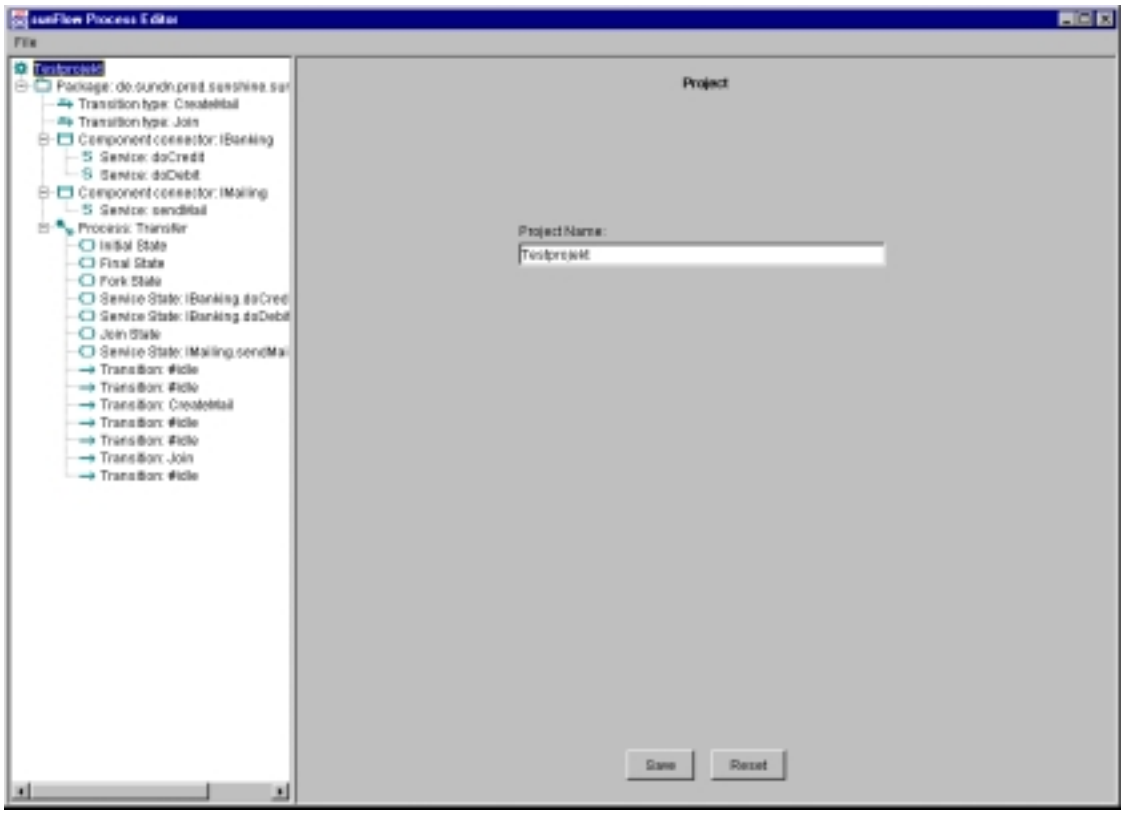


Abb. 9.1: Hauptfenster des Prozesseditors

9.2.1 Verwaltung in Projekten

Die gesamte Verwaltung der Workflow-Modelle mit Hilfe des Prozesseditors geschieht in Form von sogenannten *Projekten*, von denen jeweils höchstens eines zur gleichen Zeit im Editor geöffnet ist. Das Projekt, das zur Zeit geöffnet ist, sei der Einfachheit halber im Folgenden „das Projekt“ genannt. Wie im Abschnitt 8.3 erläutert wurde, sind alle Bestandteile der Workflow-Modelle, d.h. Konnektoren, Transitionstypen und Prozesse in Packages organisiert. Ein Projekt besteht nun aus einer Menge solcher Packages. Der Benutzer kann jederzeit Packages in das Projekt aufnehmen, die er bearbeiten möchte, oder entfernen, wenn er sie nicht bearbeiten möchte. Die Arbeit mit verschiedenen Projekten bietet den großen Vorteil, dass im Editor niemals alle Workflow-Modelle gleichzeitig sichtbar sind, in denen sich der Benutzer möglicherweise nur schwer zurecht findet, sondern immer nur diejenigen Bestandteile, die zur Zeit für seine Arbeit relevant sind. Wichtig zu erwähnen ist, dass alle Änderungen, die am Workflow-Modell vorgenommen wurden, erst dann dauerhaft übernommen werden, wenn das Projekt gespeichert wird.

9.2.2 Ansicht des Projektbaums

Im linken Teil des Editorfensters befindet sich eine hierarchische Ansicht (Baum) aller Bestandteile des Projekts. Den Wurzelknoten dieses Baums bildet das Projekt selbst. Unterhalb davon befinden sich alle Packages, die dem Projekt hinzugefügt wurden. Jedes Package enthält wiederum seine Konnektoren, Transitionstypen und Prozesse. Konnektoren haben als Unterknoten die in ihnen enthaltenen Services, Prozesse eine

Liste aller ihrer Zustände und Transitionen. Alle Aktionen, die im Editor möglich sind, werden durch Popup-Menüs aufgerufen, die angezeigt werden, wenn der Benutzer mit der rechten Maustaste einen Knoten des Projektbaums auswählt. Abhängig von der angewählten Komponente erscheinen dabei unterschiedliche Menüs.

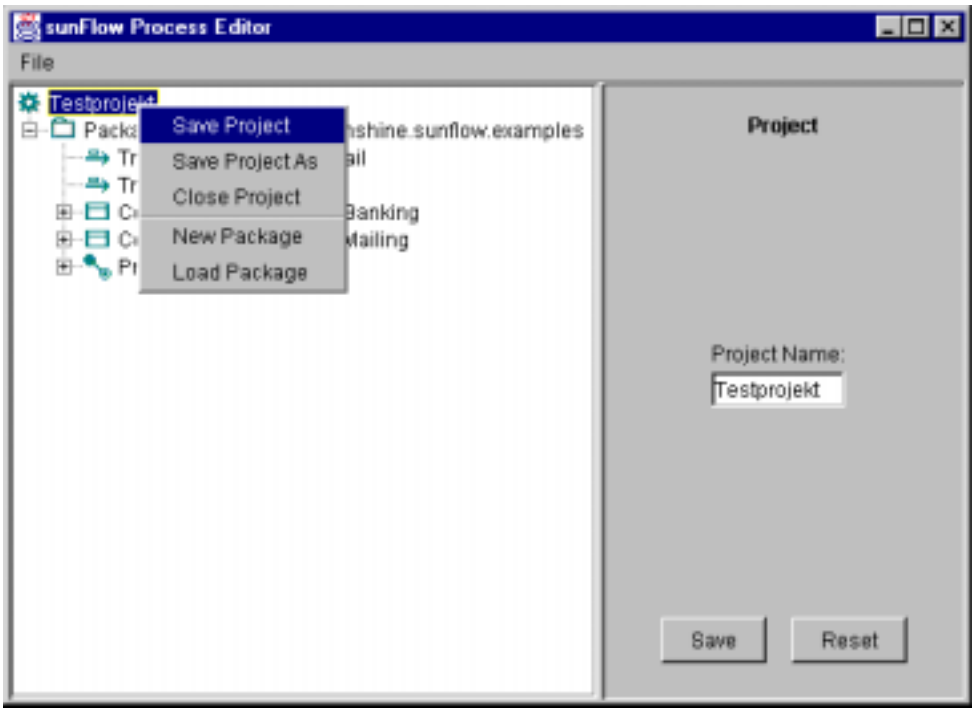


Abb. 9.2: Popup-Menüs im Projektbaum

Für den Projektknoten stehen beispielsweise folgende Aktionen zur Verfügung:

Save Project: Diese Aktion speichert alle Änderungen, die vorgenommen wurden, dauerhaft im Workflow-Modell. Falls das Projekt zuvor noch nicht mindestens einmal gespeichert wurde, erscheint ein Dateiauswahldialog. Dort kann ein Dateiname eingegeben oder eine existierende Datei ausgewählt werden, die überschrieben werden soll. Die Datei, in der das Projekt gespeichert wird, erhält automatisch die Dateiendung .sfp (sunFlow project file).

Save Project As: Es erscheint ein Dateiauswahldialog, in der ein Dateiname eingegeben oder eine existierende Datei ausgewählt werden kann, die überschrieben werden soll. Das Projekt wird dann in dieser Datei, die automatisch die Dateiendung .sfp (sunFlow project file) erhält, gespeichert.

Close Project: Schließt das Projekt. Zuvor erscheint eine Sicherheitsabfrage, ob das Projekt gespeichert werden soll.

New Package: Durch diese Aktion lässt sich ein neues Package anlegen. Es erscheint ein Eingabedialog, in dem der Name einschließlich aller übergeordneten Packages, in denen es enthalten sein soll, jeweils durch Punkt getrennt eingegeben werden muss.

Load Package: Hier lässt sich ein existierendes Package dem Projekt hinzufügen, um seine Bestandteile zu bearbeiten. Es erscheint ein Eingabedialog, in dem der Name einschließlich aller übergeordneten Packages jeweils durch Punkt getrennt eingegeben werden muss.

9.2.3 Eingabemasken

Je nachdem, welche Komponente im Projektbaum mit der Maus ausgewählt wurde, erscheint im rechten Bereich des Editors eine zugehörige Eingabemaske. In ihr werden alle Informationen, die mit der Komponente verknüpft sind, angezeigt und können dort auch direkt verändert werden. Abbildung 9.3 zeigt beispielhaft die Eingabemaske für einen Service. Alle übrigen Masken haben natürlich ein anderes Aussehen, die grundsätzliche Bedienung ist jedoch analog.

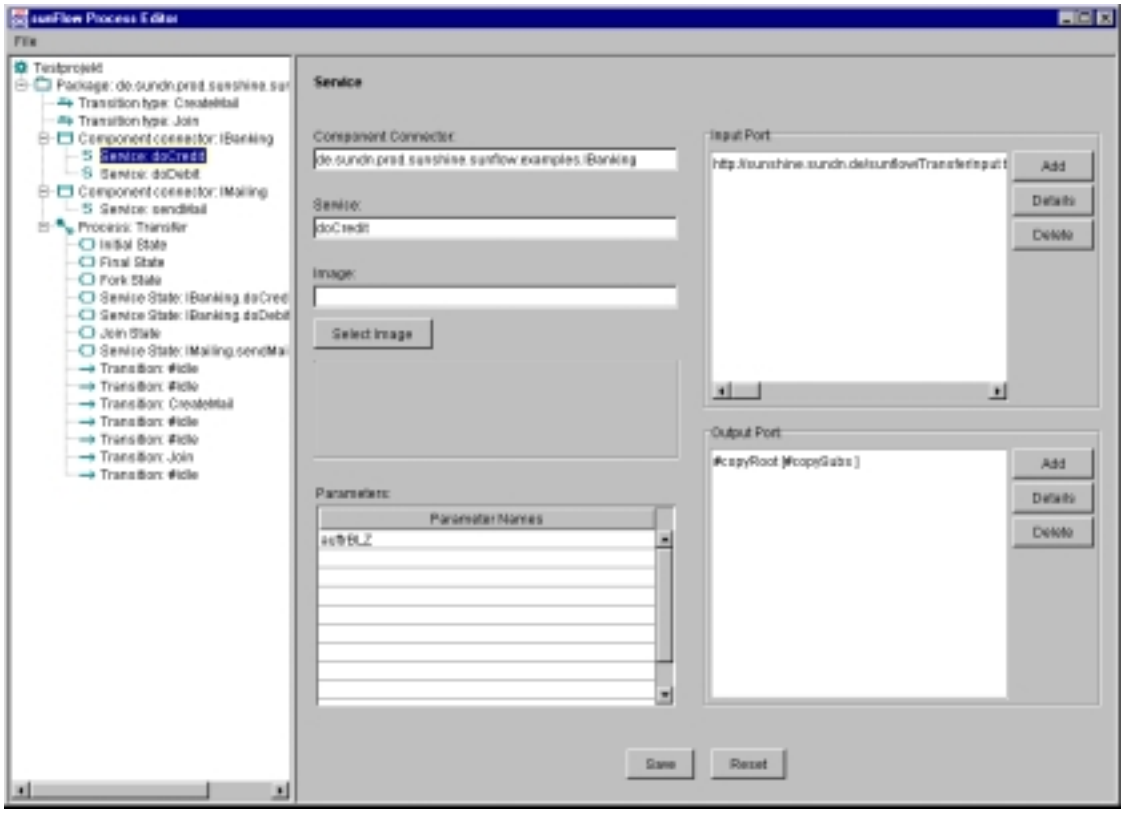


Abb. 9.3: Eingabemaske für einen Service

Links oben in der Maske wird angezeigt, dass es sich um die Eingabemaske zu einem Service handelt. Darunter ist der Name des Konnektors angegeben, zu dem der Service gehört. Diese Information kann vom Benutzer nicht verändert werden, da der Konnektor, zu dem ein Service gehört, unwiderruflich feststeht. Unterhalb davon kann bei Bedarf der Name des Service geändert werden. Des Weiteren lässt sich ein Bild auswählen, das in den Prozessdiagrammen zur Darstellung des Service verwendet wird. Entweder kann direkt in das entsprechende Feld eine URL eingegeben werden, hinter der das gewünschte Bild abgelegt ist, oder es lässt sich durch Betätigen der Schaltfläche „Select Image“ ein Bild auswählen. Es erscheint dann ein Dateiauswahldialog. Das aus-

gewählte Bild wird in der Eingabemaske angezeigt. Schließlich enthält die Maske noch eine Liste, in der die Namen der Parameter angegeben werden können, die der Service besitzt. Auf der rechten Seite befinden sich je ein Feld für den Eingangs- und Ausgangsport des Service. Zu ihrer Bedienung siehe Abschnitt 9.2.4. Jede Eingabemaske enthält ganz unten zwei Schaltflächen „Save“ und „Reset“. Beim Betätigen von „Save“ werden alle Eingaben aus der Maske in das Workflow-Modell übernommen. Mit Hilfe von „Reset“ lassen sich die vorgenommenen Änderungen zurücksetzen und der Ausgangszustand der Maske wieder herstellen. Es ist zu beachten, dass alle Änderungen verloren sind, wenn im Projektbaum eine andere Komponente ausgewählt wird, ohne vorher durch Betätigen von „Save“ die Eingaben zu speichern.

9.2.4 Eingabe von Ports

Immer wenn eine Komponente einen Port besitzt, wird dieser in der Eingabemaske durch folgendes Feld dargestellt:

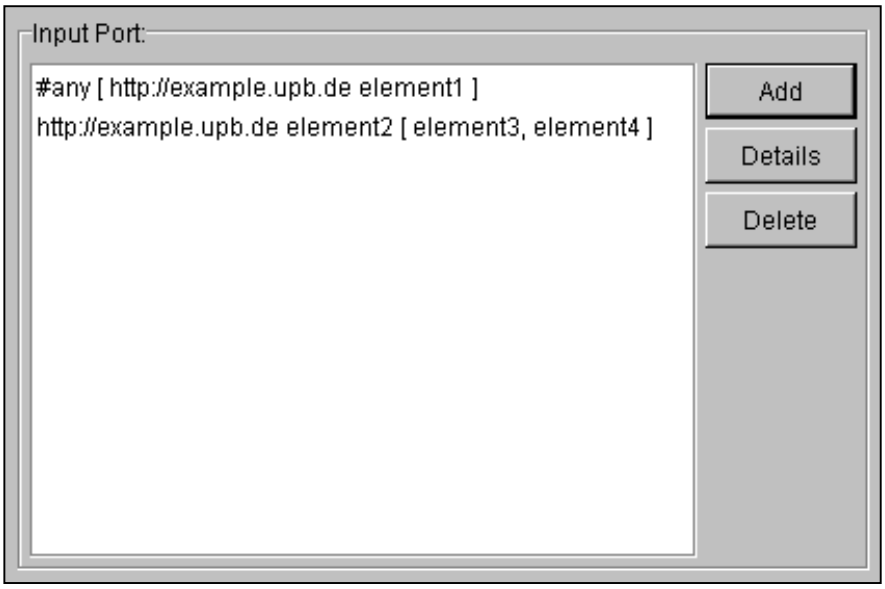


Abb. 9.4: Eingabefeld eines Ports

In der weißen Fläche sieht man eine Liste aller Dokumenttypen des Ports in der aus Abschnitt 6.3 bekannten Syntax. Ein neuer Dokumenttyp lässt sich hinzufügen, indem man die Schaltfläche „Add“ auswählt. Es erscheint dann der in Abbildung 9.5 dargestellte Eingabedialog. Wenn in der Liste ein Dokumenttyp mit der Maus ausgewählt wurde, lässt er sich entweder durch „Details“ bearbeiten - es erscheint dann ebenfalls der Dialog aus Abbildung 9.4 - oder durch „Delete“ aus dem Port entfernen. Ein Doppelklick auf einen Dokumenttyp hat denselben Effekt, wie ein Betätigen von „Details“.

Um einen Dokumenttyp zu erzeugen oder zu bearbeiten, dient der modale Dialog aus Abbildung 9.5. Ganz links im Dialog lässt sich auswählen, ob der Dokumenttyp ein konkretes Wurzelelement oder den Platzhalter #any enthalten soll. Falls es sich um einen offenen Port handelt, steht außerdem #copyRoot zur Auswahl. In der Tabelle rechts kann eine Menge von Unterelementen angegeben werden. Genau wie für das Wurzelelement, müssen dabei sowohl der Namensraum als auch der Name des XML-Element eingegeben werden. Für #copyRoot lässt sich eine Menge von Ausnahmen

angeben (siehe Abschnitt 6.3.1). Des Weiteren kann durch Markieren des entsprechenden Feldes bestimmt werden, dass auch die Unterelemente übernommen werden (#copySubs), wobei sich ebenfalls Ausnahmen spezifizieren lassen. Der Dialog kann mit „OK“ oder „Cancel“ beendet werden. Im Fall von „OK“ werden die Eingabe in das Feld des zugehörigen Ports in der Eingabemaske übernommen, bei „Cancel“ werden die Eingaben verworfen. Zu beachten ist, dass die Änderungen am Port erst dann in das Workflow-Modell übernommen werden, wenn in der Eingabemaske der Komponente die Schalfläche „Save“ betätigt wird. Es genügt nicht, den Dialog des Dokumenttyps mit „OK“ zu beenden.

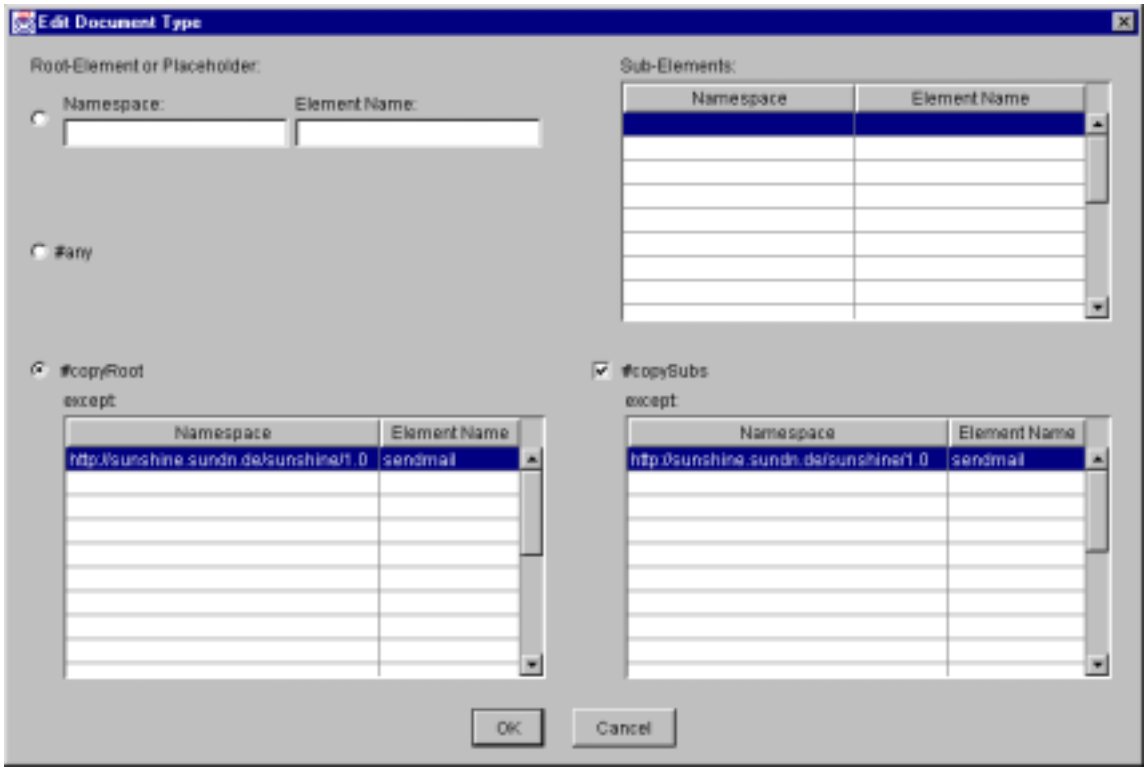


Abb. 9.5: Eingabedialog eines Dokumenttyps

9.2.5 Konstruieren eines Prozesses

Im vorausgegangenen Abschnitt wurden die einzelnen Teile der Benutzungsoberfläche und grundlegende Bedienungstechniken erklärt. Darauf aufbauend soll nun hier in einer Art Tutorial schrittweise erläutert werden, wie sich mit Hilfe des Prozesseditors ein Workflow-Modell entwerfen lässt. Dazu muss zunächst der Editor gestartet, und das gewünschte Projekt geladen oder neu angelegt werden (siehe Abschnitt 9.2.1).

Erzeugen neuer Konnektoren

Wie bereits mehrfach erwähnt, bestehen die statischen Bausteine von Prozessdiagrammen aus Konnektoren und Transitionstypen. Diese müssen entweder vor der Modellierung des Prozesses oder bei Bedarf definiert werden, sofern nicht schon alle für den Prozess benötigten Komponenten existieren. Falls sich das Package, in dem ein neuer Konnektor oder Transitionstyp enthalten sein soll, noch nicht im Projekt befindet, muss

es zunächst geladen werden (siehe Abschnitt 9.2.2).

Um einen neuen Konnektor anzulegen, betätigt man mit der rechten Maustaste das Package im Projektbaum, das den neuen Konnektor enthalten soll. Es erscheint ein Popup-Menü, in dem die Aktion „New Connector“ gewählt werden muss (siehe Abbildung 9.6). Der Editor legt dann einen neuen Konnektor an, der sofort im Baum sichtbar wird und bereits selektiert ist. Im rechten Bereich des Fensters sieht man daher die zugehörige Eingabemaske, in der man insbesondere den gewünschten Namen des Konnektors eingeben kann.

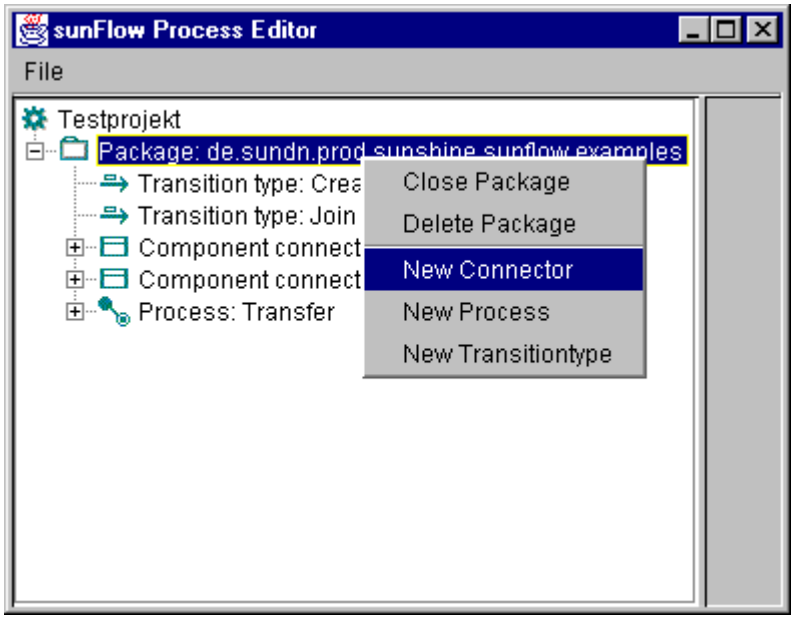


Abb. 9.6: Erzeugen eines neuen Konnektors

Durch Auswahl des Konnektors im Baum und Auswahl der Aktion „New Service“ im dann erscheinenden Popup-Menü lassen sich Services zum Konnektor hinzufügen. Jeder Service wird dabei in der unter 9.2.3 beschriebenen Eingabemaske definiert.

Erzeugen neuer Transitionstypen

Analog zum Anlegen eines Konnektors wählt man mit der rechten Maustaste das Package im Projektbaum aus, das den neuen Transitionstyp enthalten soll; es erscheint ein Popup-Menü. Nach Auswahl der Aktion „New Transitiontype“ legt der Editor einen neuen Typ an, der sofort im Baum sichtbar wird und selektiert ist. Im rechten Bereich des Fensters sieht man dann die in Abbildung 9.7 gezeigte Eingabemaske. Dort lässt sich oben links der gewünschte Name des Typs angeben. Darunter wird das Package angezeigt, in dem sich der Transitionstyp befindet; dies kann bei Bedarf durch Betätigen der Schaltfläche „Move“ geändert werden. Es erscheint dann ein Dialog, in dem sich eines der geladenen Packages auswählen lässt, in das der Typ verschoben werden soll.

Darunter kann man ein Stylesheet definieren, mit dem Migrationsdokumente transformiert werden sollen, wenn sie über eine Transition dieses Typs laufen. Es kann entweder eine URL für ein Stylesheet angegeben, eine Datei mit Hilfe der Schaltfläche „Select File“ ausgewählt oder ein Stylesheet direkt in der Eingabemaske eingegeben werden. Im letzten Fall prüft der Editor beim Betätigen von „Save“ automatisch, ob die

Benutzereingabe ein gültiges XSL-Stylesheet enthält. Falls nicht, werden die gefundenen Fehler in einer Fehlermeldung ausgegeben.

Auf der rechten Seite befinden sich die aus 9.2.4 bekannten Felder zur Eingabe des Quell- und Zielports des Transitionstyps.

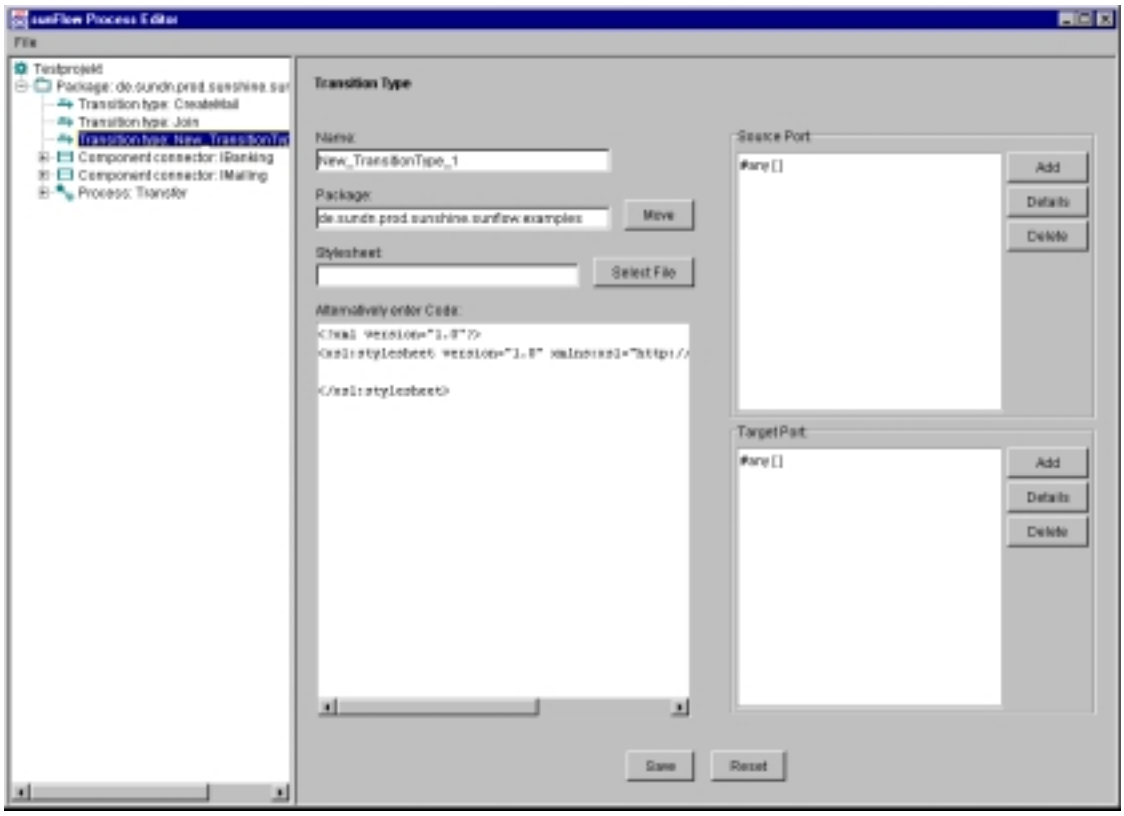


Abb. 9.7: Eingabemaske für einen neuen Transitionstyp

Erzeugen eines neuen Prozessdiagramms

Um ein neues Prozessdiagramm anzulegen, wählt man mit der rechten Maustaste das Package im Projektbaum aus, das den neuen Prozess enthalten soll. Es erscheint ein Popup-Menü, in dem die Aktion „New Process“ gewählt werden muss. Der Editor legt dann einen neuen Prozess an, der sofort im Baum sichtbar wird und bereits selektiert ist. Im rechten Bereich des Fensters erscheint die zugehörige Eingabemaske, die aus zwei Karteireitern besteht (siehe Abbildung 9.8).

Im Reiter „Properties“ kann man die Eigenschaften des Prozesses einstellen: Auf der linken Seite lässt sich der Name des Prozesses und bei Bedarf sein Package verändern. Außerdem kann festgelegt werden, ob er transaktional ausgeführt werden soll oder nicht. Auf der rechten Seite befinden sich die Felder für die Ports, wobei sich jedoch nur der Eingangsport editieren lässt. Der Ausgangsport wird automatisch vom Editor anhand der Prozessstruktur berechnet (vgl. Abschnitt 7.6).

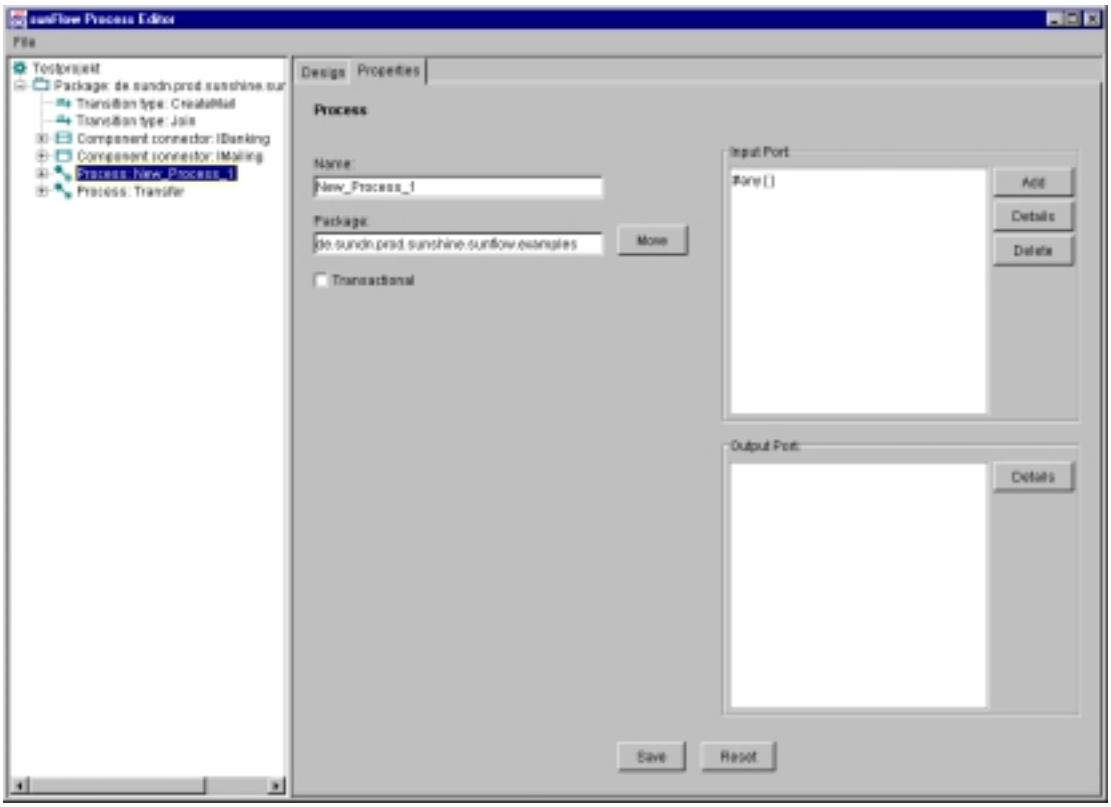


Abb. 9.8: Eingabemaske für einen neuen Prozess

Um den Prozess selbst mit seinen Zuständen und Transitionen zu modellieren, wechselt man in den Karteireiter „Design“, in dem bereits je ein Start- und Endzustand existiert.

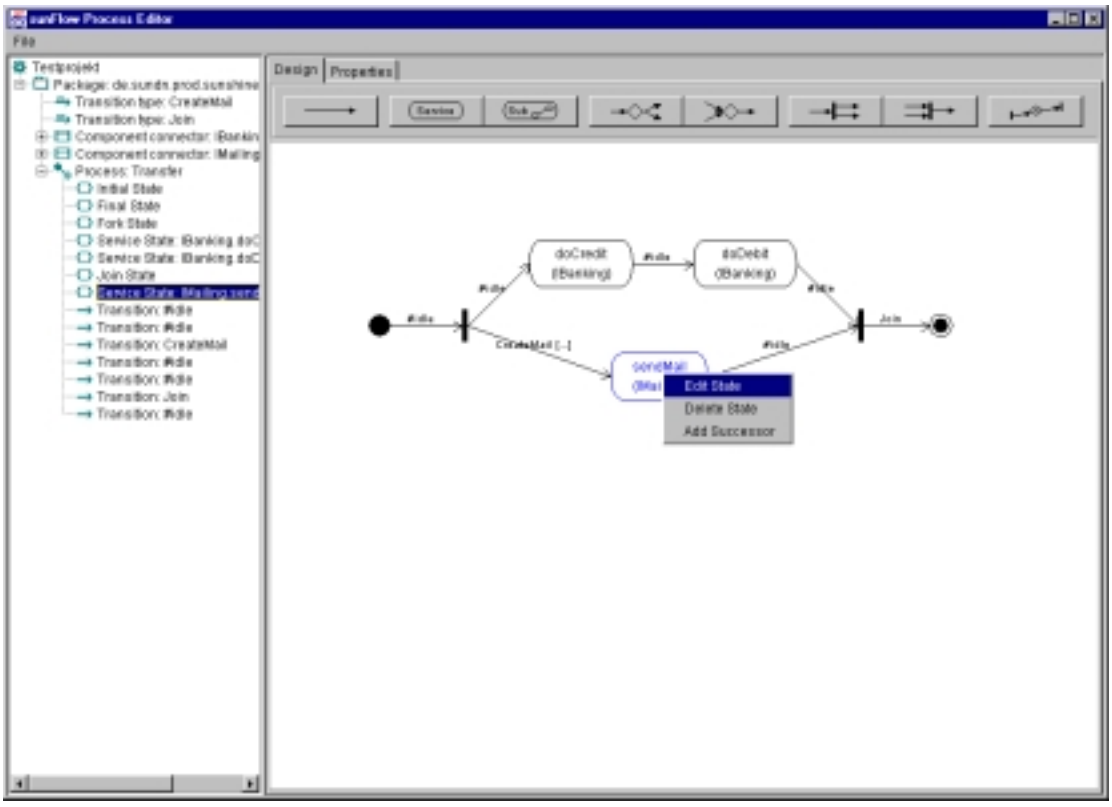


Abb. 9.9: Zeichenfläche für ein Prozessdiagramm

Der Karteireiter „Design“ besteht aus einer großen Zeichenfläche, in dem der ausgewählte Prozess grafisch in Form eines Prozessdiagramms dargestellt ist. Alle grafischen Elemente des Diagramms, d.h. seine Zustände und Transitionen, lassen sich mit der rechten Maustaste auswählen. Ähnlich zum Projektbaum erscheint dann ein Popup-Menü, in dem die möglichen Aktionen angeboten werden. Alle Elemente lassen sich außerdem bei gedrückter linker Maustaste beliebig verschieben und neu positionieren.

Hinzufügen von Zuständen

Einem Prozessdiagramm lassen sich bei Bedarf neue Zustände hinzufügen. Dazu dienen die Schaltflächen oben in der Zeichenfläche. Dort finden sich Schaltflächen zur Erzeugung von Service-, Subactivity-, Fork-, Join-, Decision- und Merge-Zuständen. Eine Besonderheit stellen Synchronisationszustände dar. Mit Hilfe der Schaltfläche ganz rechts fügt man eine komplette Synchronisationsstelle, bestehend aus Synchfork-, Synchjoin- und Synch-Zustand in das Prozessdiagramm ein. Diese lassen sich auch nur als Ganzes wieder löschen, da sie ausschließlich in dieser Konstellation erlaubt sind.

Nach dem Betätigen einer Schaltfläche wird der gewünschte Zustand oben links in der Zeichenfläche eingefügt; er kann dann mit der Maus an die gewünschte Stelle verschoben werden. Um seine Eigenschaften anzusehen bzw. zu editieren, muss er in der Zeichenfläche oder im Projektbaum selektiert werden. Wenn dann auf den Karteireiter „Properties“ umgeschaltet wird, erscheint die passende Eingabemaske. Abbildung 9.10 zeigt die Maske eines Service-Zustands:

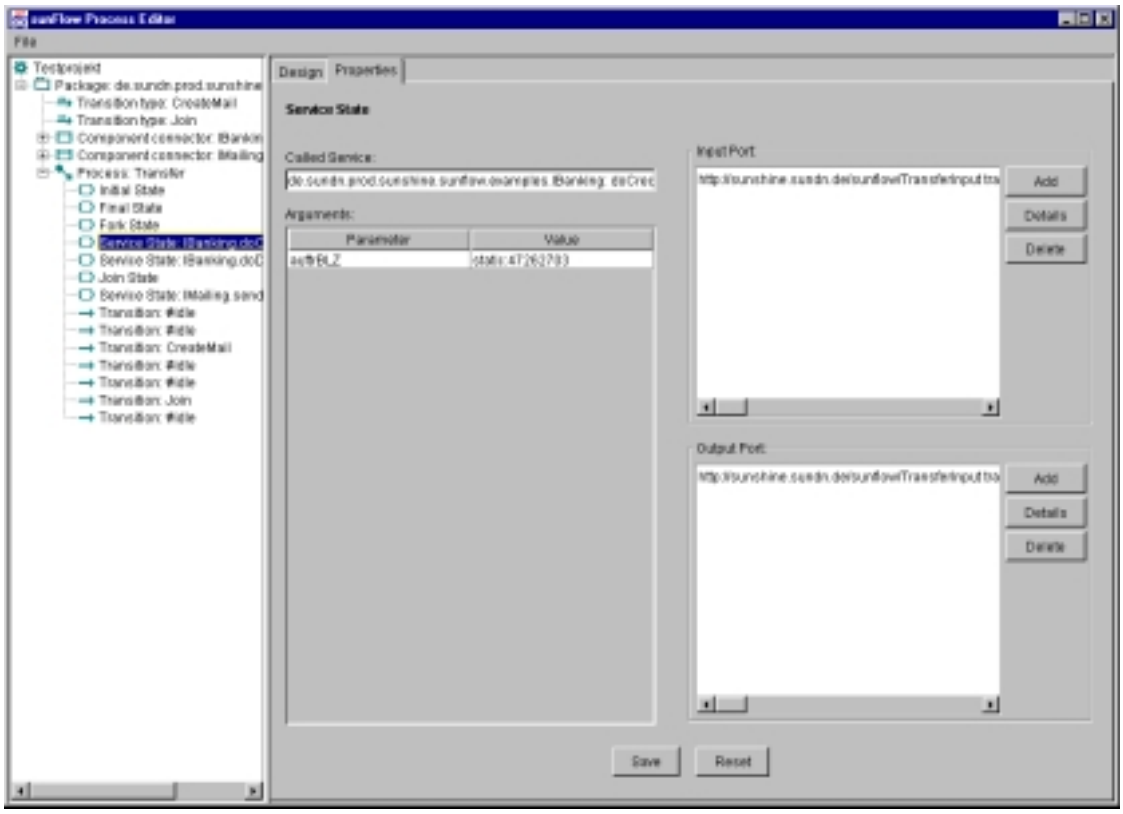


Abb. 9.10: Eingabemaske eines Service-Zustands

Zu beachten ist, dass nach dem Betätigen der Schaltflächen für Service- und Subaktivitätszustand nicht unmittelbar die grafischen Elemente in das Prozessdiagramm einge-

fügt werden. Zuvor erscheint jeweils ein modaler Dialog, in dem der gewünschte Service bzw. Subprozess ausgewählt werden muss.

Für Service-Zustände existiert eine weitere Besonderheit. Beim Erzeugen eines solchen Zustands werden zunächst die Ports des Service aus dem statischen Modell übernommen. Der Modellierer kann jedoch in der Eingabemaske die Ports manuell anpassen, wenn er zum Zeitpunkt der Prozessmodellierung mehr Informationen über die Dokumentstrukturen besitzt, als bei der Definition des statischen Modells (vgl. Abschnitt 6.3.1, Einbindung von Ports in Prozessdiagramme). Zu beachten ist dabei jedoch, dass die Teilportbedingungen aus 6.3.1 nicht verletzt werden dürfen. Der Editor gibt in diesem Fall eine Fehlermeldung aus.

Hinzufügen von Transitionen

Zustände, die nach dem weiter oben beschriebenen Verfahren angelegt wurden, müssen mit Hilfe von Transitionen mit dem restlichen Prozessdiagramm verbunden werden, damit das Diagramm einen Sinn ergibt. Da eine Transition immer genau zwei Zustände verbindet, müssen auch in der Zeichenfläche zwei Zustände selektiert werden, um sie zu verbinden. Zunächst muss durch Drücken der linken Maustaste der Zustand selektiert werden, von dem die Transition ausgehen soll. Anschließend wird der Zielzustand der Transition bestimmt, indem er bei gedrückter „Shift“- oder „Strg“-Taste mit der linken Maustaste angewählt wird. Nun muss die Schaltfläche links oben in der Zeichenfläche betätigt werden. Es erscheint dann der folgende modale Dialog:

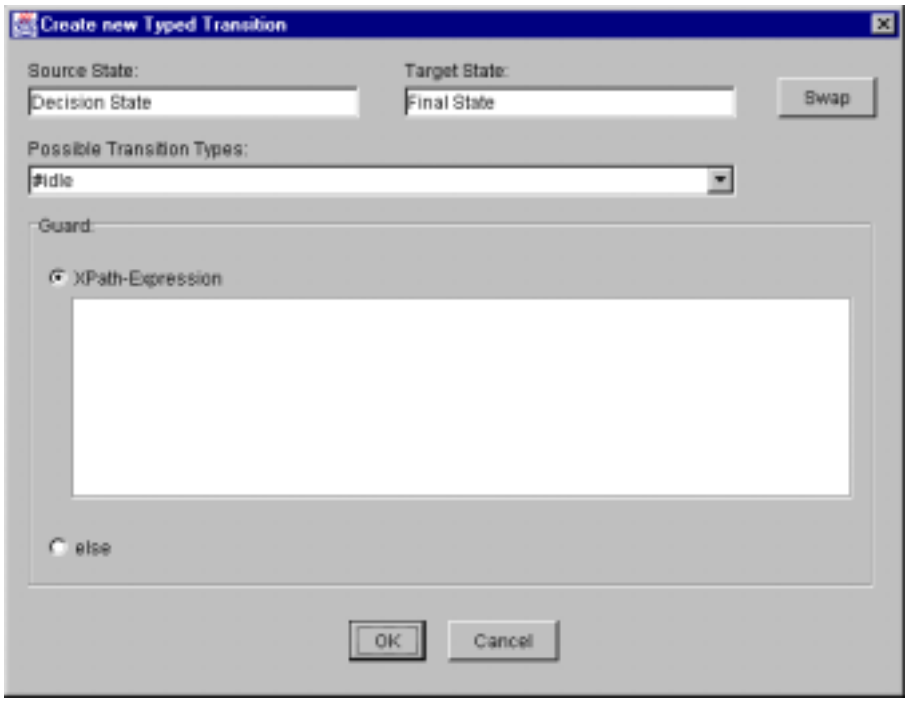


Abb. 9.11: Eingabedialog zum Erzeugen einer Transition

Oben im Dialog wird der Quell- und Zielzustand der anzulegenden Transition angezeigt. Die beiden Zustände lassen sich mit Hilfe von „Swap“ vertauschen, falls sie in der falschen Reihenfolge markiert wurden. Darunter muss der gewünschte Transitionstyp ausgewählt werden. Durch das Konzept der Ports ist es dem Editor hier möglich, zu prüfen, welche Typen an dieser Stelle des Prozesses überhaupt einsetzbar sind. Nur sol-

che werden in die Auswahlliste des Dialogs aufgenommen und stehen daher zur Verfügung. Eingesetzt werden dürfen Typen genau dann, wenn die resultierende Transition eine Transition im Sinne von Def. 6.17 (siehe Abschnitt 6.3.3) ist. Der Migrationssatz (6.18, Abschnitt 6.3.3) besagt, dass in diesem Fall die Verwendung des jeweiligen Transitionstyps zu einer konsistenten Portverkettung führt. Es kann auch passieren, dass die beiden ausgewählten Zustände sich überhaupt nicht verbinden lassen. In diesem Fall erscheint im Editor eine entsprechende Meldung. Der Benutzer muss dann zunächst einen geeigneten Transitionstyp erzeugen, dessen Stylesheet das Migrationsdokument so transformiert, dass eine Verbindung der Zustände möglich wird. Dieses Vorgehen bietet den Vorteil, dass Inkonsistenzen im Prozessdiagramm von vornherein vermieden werden, indem stets darauf geachtet wird, dass nur Ports verkettet werden, die die Teilport-Bedingung erfüllen. Aus dem Kompatibilitätssatz (6.10, Abschnitt 6.3.3) folgt dann, dass die Verarbeitung des Dokuments durch die Services des Diagramms konsistent ist. Zu beachten ist, dass im Editor aus Gründen der Übersichtlichkeit nur die Transitionstypen zur Verfügung stehen, deren Package sich im Projekt befindet. Falls ein anderer Typ verwendet werden soll, muss zunächst sein Package in das Projekt aufgenommen werden.

Für den Guard einer Transition sind drei Fälle zu unterscheiden:

1. Wenn der Quellzustand ein Fork-Zustand ist, kann für die Transition ein Guard angeben werden. Im Dialog kann man dann im Feld „XPath-Expression“ den gewünschten XPath-Ausdruck angeben.
2. Wenn der Quellzustand ein Decision-Zustand ist, muss für die Transition ein Guard angeben werden. Im Dialog muss man entweder im Feld „XPath-Expression“ den gewünschten XPath-Ausdruck angeben oder „else“ auswählen, um zu signalisieren, dass bei der Ausführung dieser Weg genommen werden soll, falls alle anderen Guards nicht erfüllt sind.
3. In allen anderen Fällen ist kein Guard erlaubt. Das zugehörige Feld im Dialog ist daher deaktiviert.

Folgezustand bestimmen

Alternativ zu dem bisher beschriebenen Vorgehen, bei dem zunächst einzelne Zustände in ein Prozessdiagramm eingefügt werden, die anschließend mit passenden Transitionen verbunden werden, bietet der Prozesseditor auch die Möglichkeit, direkt zu jedem Zustand einen Nachfolgezustand zu erzeugen. Dazu wählt man den gewünschten Zustand mit der rechten Maustaste an und ruft die Aktion „Add Successor“ auf. Es erscheint ein modaler Dialog mit zwei Karteireitern, einer zum Anlegen eines Service- und der andere zum Anlegen eines Subaktivitätszustands. Anhand der Ports überprüft der Editor, welche Services und Prozesse an dieser Stelle des Prozesses aufgerufen werden können. Dabei werden auch die Transitionstypen berücksichtigt, die aufgrund ihrer Stylesheets den Aufruf von Services und Prozessen ermöglichen, deren Port eigentlich nicht kompatibel zum Ausgangsport des gewählten Zustands sind. Der Prozesseditor bietet also nur solche Services und Prozesse an, deren Verwendung die Bedingungen des Kompatibilitätssatzes und des Migrationssatzes (siehe Abschnitt 6.3.3) erfüllen, so dass sich ein konsistenter Dokumentfluss durch das Diagramm ergibt. Der Benutzer kann sich aus der so generierten Liste eine Kombination aus Transitionstyp und Service oder Prozess aussuchen, die dann im Diagramm erzeugt wird.

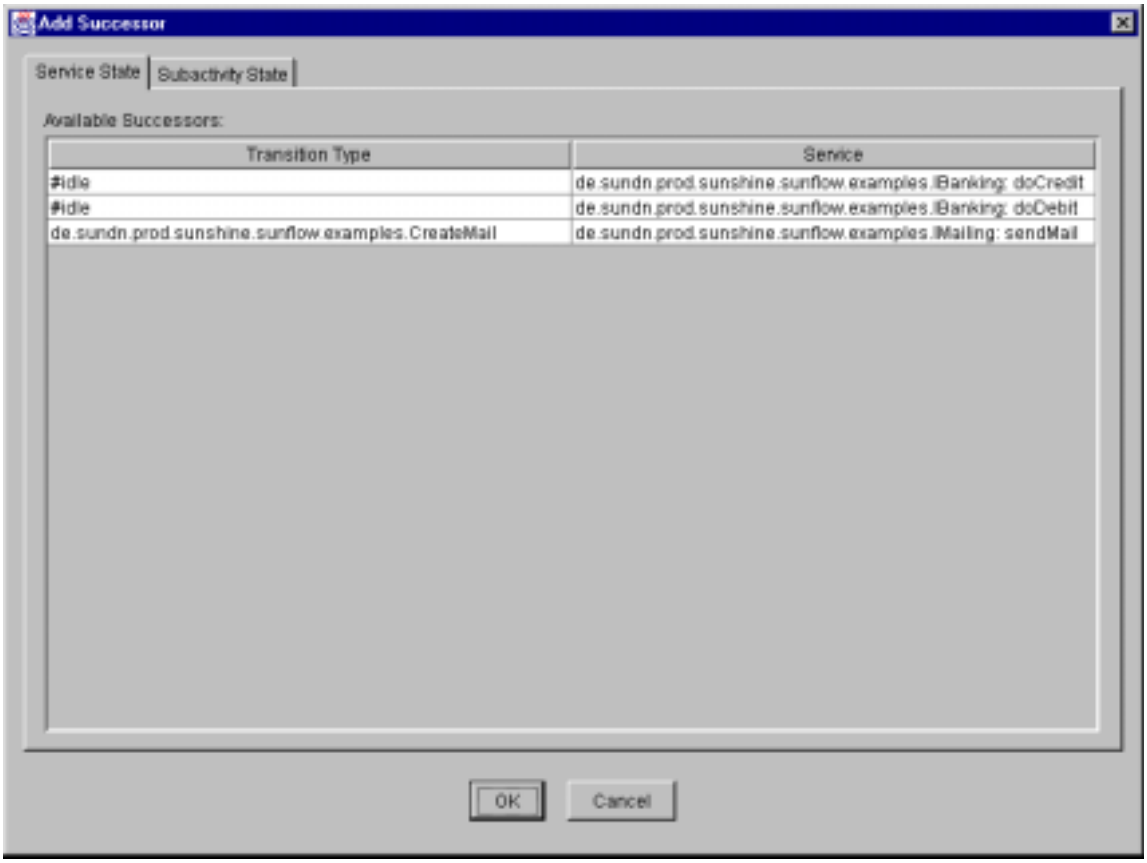


Abb. 9.12: Eingabedialog zum Erzeugen eines Folgezustands

Das sukzessive Anwenden dieser Aktion ausgehend vom Startzustand erlaubt es dem Benutzer, auf einfache und komfortable Weise konsistente Prozessmodelle zu entwerfen. Besonders wenn nach einiger Zeit eine größere Menge von vordefinierten Services, Transitionstypen und (Sub-)prozessen zur Verfügung steht, dürfte die Aktion ein sehr effizientes Modellieren erlauben.

Validieren von Prozessdiagrammen

Der Benutzer kann jederzeit überprüfen, ob sein Workflow-Modell ein gültiges Prozessdiagramm darstellt. Dazu wählt er mit der rechten Maustaste den gewünschten Prozess im Projektbaum aus und ruft im Popup-Menü die Aktion „Validate Process“ auf. Der Prozesseditor überprüft das Diagramm dann auf jede Art von Fehlern und Inkonsistenzen. Dazu verwendet er den im Abschnitt 7.5 beschriebenen Validierungsalgorithmus. Anschließend erscheint ein Ergebnisdialog, in dem sich ggf. Meldungen über gefundene Fehler befinden. Die Validierung bricht nicht beim ersten Fehler ab, sondern versucht, so viele Inkonsistenzen wie möglich zu finden. Dabei kann es passieren, dass sogenannte Folgefehler auftreten, oder in Ausnahmefällen Meldungen produziert werden, die nur schwer nachvollziehbar sind. Sobald die Ursache des Hauptfehlers behoben ist, verschwinden beim nächsten Validierungsvorgang auch diese Meldungen.

Eine Validierung aller im Projekt enthaltenen Prozesse wird automatisch durchgeführt, bevor das Projekt gespeichert wird. Das stellt sicher, dass niemals Projekte mit inkonsistenten Workflow-Modellen dauerhaft abgelegt werden.

9.3 Implementierung des Prozesseditors

In diesem Abschnitt sollen nun wichtige Aspekte der Implementierung des Prozesseditors betrachtet werden. Der Editor wurde von uns in der Sprache Java realisiert, wobei insbesondere die Java-Swing-Bibliotheken zum Einsatz kamen, die die notwendige Funktionalität zur Verfügung stellen, um grafische Benutzungsoberflächen zu erstellen.

9.3.1 Klasse AppFrame

Die wichtigste Klasse des Prozesseditor ist *AppFrame*. Dabei handelt es sich um die Klasse, die das Hauptfenster repräsentiert (vgl. Abschnitt 9.2). Sie ist innerhalb der Implementierung zugleich der Einstiegspunkt des Systems, über den alle wichtigen Bestandteile erreichbar sind. Aufgrund dieser Aufgabe haben wir uns entschieden, die Klasse gemäß dem Entwurfsmuster *Singleton* (siehe [Gam96] S.157) zu entwerfen. Es kann damit sichergestellt werden, dass von der entsprechenden Klasse maximal eine Instanz existiert und diese zudem von überall her zugänglich ist, ohne möglicherweise über eine Vielzahl von Assoziationen zu navigieren, um auf sie zugreifen zu können. Abbildung 9.13 zeigt die Klasse und ihre wichtigsten Bestandteile:

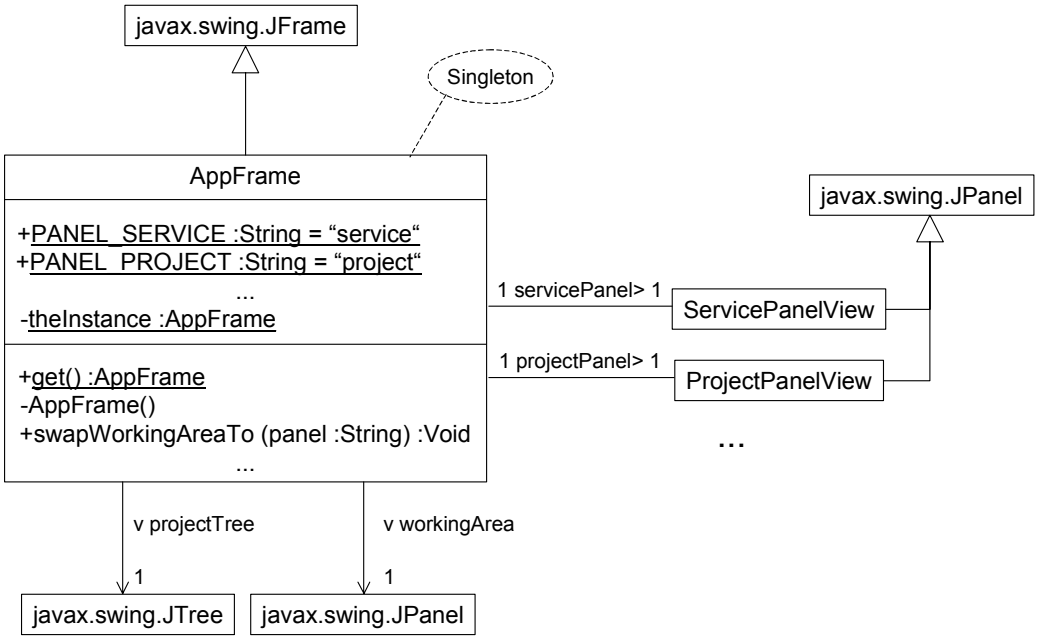


Abb. 9.13: Klasse AppFrame

AppFrame erbt von der Swing-Klasse JFrame, damit sie als Repräsentation des Editor-Hauptfensters verwendet werden kann. Sie hat entsprechend der Beschreibung aus Abschnitt 9.2 Referenzen zu ihren zwei Bestandteilen, dem Projektbaum (projectTree) und dem Bereich, in den die Eingabemasken eingeblendet werden (workingArea).

Das Attribut `theInstance` und die Methode `get()`, die beide Klasseneigenschaften (in Java: statisch) sind, realisieren in Kombination mit dem Konstruktor, dem die Sichtbarkeit „private“ zugeordnet ist, die Eigenschaft der Klasse als Singleton. Die einzige Objektinstanz der Klasse ist im Attribut `theInstance` gespeichert und man erhält Zugriff darauf, indem man die Methode `get()` aufruft. Das direkte Erzeugen einer Instanz mittels Konstruktoraufruf ist von außen hingegen nicht möglich.

Weiterhin besitzt die Klasse AppFrame zu jedem Knotentyp, der im Projektbaum vorkommen kann, genau eine Instanz der zugehörigen Eingabemaske. Dazu zählt beispielsweise eine Maske für Services (ServicePanelView), eine für das Projekt (ProjectPanelView) usw. Immer wenn im Baum ein Knoten markiert wird, wird die passende Maske in den rechten Bereich des Fensters eingeblendet und die Daten des Knotens als aktuelle Werte in die Maske eingetragen. Das Einblenden einer Maske im Fenster geschieht mit Hilfe der Methode `swapWorkingAreaTo(..)`. Um die gewünschte Maske auszuwählen, existieren die vordefinierten Konstanten PANEL_..., von denen man die passende als Parameter an die Methode übergibt.

9.3.2 Anwendung des Model-View-Controller-Konzepts

Für die Realisierung der Benutzungsoberfläche und ihr Verhalten haben wir das sogenannte Model-View-Controller-Konzept (MVC) eingesetzt (siehe [Gam96] S.5). Die Grundidee dieses Ansatzes ist es, in Softwaresystemen drei Aspekte voneinander zu trennen: Die Daten und Funktionen selbst, auf denen das System arbeitet, ihre Darstellung in der Benutzungsoberfläche und die Programmlogik, die auf Eingaben des Benutzers bzw. Ereignisse in der Oberfläche reagiert. In unserem Fall besteht das *Model* aus den Klassen des Datenmodells (siehe Abschnitt 7.4), die *Views* aus den Eingabemasken und die *Controller* aus sogenannten ActionListener-Klassen, die die Benutzeraktionen auswerten. Für jeden Knotentyp des Projektbaums gibt es eine „MVC-Einheit“. Im Folgenden soll die Umsetzung des Konzepts für Services erläutert werden; für alle anderen Datenklassen gibt es analoge Bestandteile im Prozesseditor, so dass diese Klassenstruktur das zentrale Muster innerhalb des Editors ist. Abbildung 9.14 zeigt die Klassen zur Verwaltung eines Service:

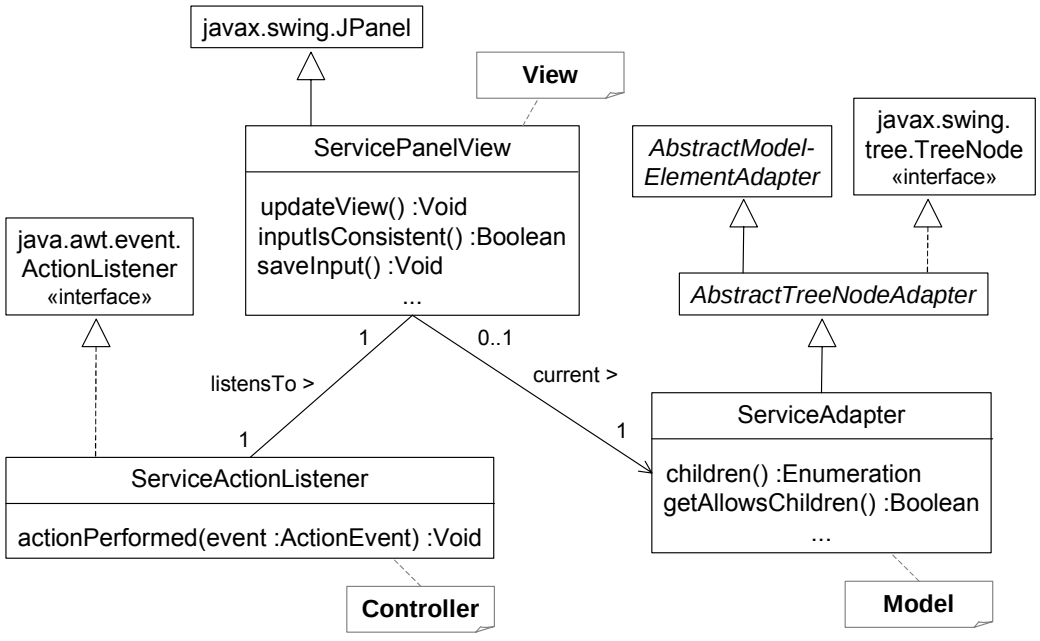


Abb. 9.14: MVC-Klassen für Services

Model

Das Model besteht aus der Klasse *ServiceAdapter*, die einen Service repräsentiert. Es kann hier nicht direkt die Klasse Service aus dem Datenmodell verwendet werden, da der Editor zusätzliche Funktionalität benötigt. Daher wird mit Hilfe des in Abschnitt 7.4 (Absatz: Flexible Zugriffsschicht auf Elemente des Datenmodells) beschriebenen Adapter-Mechanismus die Schnittstelle der Klasse Service erweitert. Zu diesem Zweck erbt ServiceAdapter indirekt von der Klasse AbstactModelElementAdapter. Zusätzlich erfüllt sie die vom Interface TreeNode geforderte Schnittstelle, die aus Methoden wie `children()`, `getAllowsChildren()` usw. besteht. Dieses Vorgehen erlaubt es, Objekte der Klasse ServiceAdapter direkt als Knoten in den Projektbaum des Hauptfensters einzufügen. Die Methoden werden vom Swing-System dazu benutzt, die grafische Darstellung des Baums zu generieren.

View

Die Darstellung eines Service geschieht mit Hilfe einer Eingabemaske, die durch die Klasse *ServicePanelView* implementiert wird. Sie ist von der Swing-Klasse JPanel abgeleitet, die die notwendigen Methoden anbietet, um eine solche Maske zu erstellen. Im Konstruktor von ServicePanelView werden alle benötigten grafischen Elemente wie Eingabefelder, Listen, Schaltflächen usw. erzeugt.

Es gibt prinzipiell zwei verschiedene Möglichkeiten, die Darstellung eines Modellelements zu realisieren. Zum einen könnte man für jeden konkreten Service eine komplette MVC-Einheit im Speicher anlegen. D.h. jeder Service hätte seine eigene Eingabemaske, die in das Hauptfenster eingeblendet wird, sobald der Service im Projektbaum selektiert wird. Da jedoch die Klassen des Java-Swing-Systems recht umfangreich sind, würde dadurch ein hoher Speicherbedarf des Prozesseditors entstehen, was besonders bei komplexen Workflow-Modellen Probleme bereiten könnte. Wir haben uns aus diesem Grund gegen diese Variante entschieden. Stattdessen besitzt die Klasse AppFrame, wie bereits in 9.3.1 erwähnt, genau eine Eingabemaske, die von allen Services genutzt wird. In Abbildung 9.14 sieht man daher, dass zwar eine Instanz von ServicePanelView immer eine Assoziation zu einem ServiceAdapter besitzt (current), umgekehrt aber nicht jeder ServiceAdapter in einer Maske angezeigt wird. Nachdem diese Current-Assoziation verändert wurde, da der Benutzer einen anderen Service ausgewählt hat, erfolgt ein Aufruf der Methode `updateView()`. Sie ist dafür zuständig, die Daten aus dem neuen Service auszulesen und in die grafischen Bedienelemente der Maske einzutragen.

Controller

Damit Aktionen, die der Benutzer in der Oberfläche ausführt, vom Prozesseditor verarbeitet werden können, besitzt ServicePanelView eine Assoziation zur Klasse *ServiceActionListener*. Diese erfüllt das Swing-Interface ActionListener und implementiert seine Methode `actionPerformed(..)`. Die Methode wird vom Swing-System aufgerufen, sobald eine Benutzeraktion erfolgt. Innerhalb der Methode muss dann anhand des Event-Objekts, das als Parameter übergeben wurde, festgestellt werden, um welche Aktion es sich handelt. In den Eingabemasken des Editors sind zwei Ereignisse relevant, das Betätigen der „Save“- und der „Reset“-Schaltfläche (vgl. Abschnitt 9.2.3). Im Falle von Reset kann einfach die Methode `updateView()` aus ServicePanelView aufgerufen werden, da sie die Daten des Service neu aus dem Datenmodell liest und

etwaige Benutzereingaben in der Maske überschreibt. Im Falle von Save hingegen erfolgt ein Aufruf von `saveInput()`, um die Eingaben ins Datenmodell zu übernehmen. Zuvor findet eine Überprüfung aller Benutzereingaben mittels `inputIsConsistent()` statt. Nur wenn keine Inkonsistenzen gefunden wurden, können die Daten gespeichert werden, ansonsten erscheint eine entsprechende Fehlermeldung, die den Benutzer auf Eingabefehler hinweist.

9.3.3 Die Prozesszeichenfläche

Im Abschnitt 9.2.5 wurde die Prozesszeichenfläche beschrieben, in der alle Zustände und Transitionen eines Prozessdiagramms grafisch dargestellt werden. Zustände können zudem mit der Maus beliebig verschoben werden. Im Sinne des MVC-Konzepts handelt es sich bei der grafischen Darstellung der Modellelemente um eine zusätzliche View-Komponente neben der Eingabemaske, so dass solche Elemente also zwei verschiedene Repräsentationen in der Benutzungsoberfläche haben. Abbildung 9.15 zeigt die notwendigen Klassen am Beispiel eines Service-Zustands:

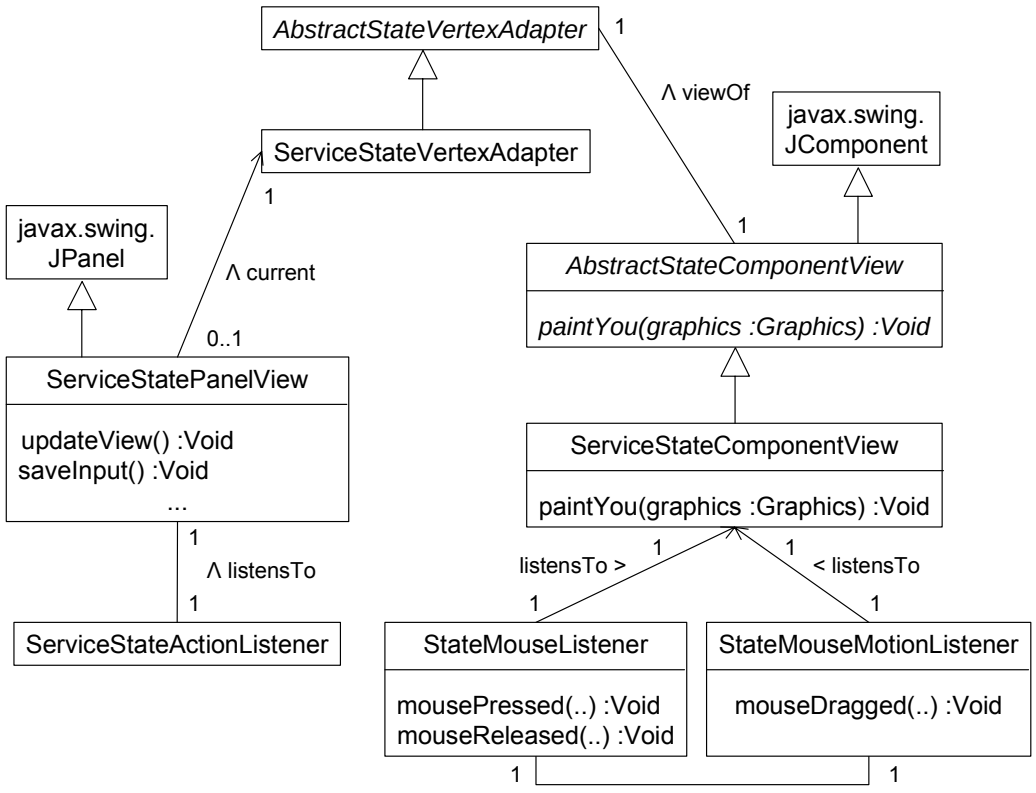


Abb. 9.15: MVC-Klassen für Service-Zustände

Auf der linken Seite der Abbildung sieht man die Klassen, die die Eingabemaske eines Service-Zustands realisieren. Die analogen Klassen für einen Service wurden bereits im Abschnitt 9.3.2 beschrieben. Daneben existiert nun als zweite View-Komponente die Klasse *ServiceStateComponentView*. Sie erbt indirekt von der Swing-Klasse JComponent, die die notwendigen Methoden enthält, um ein grafisch darstellbares Element der Benutzungsoberfläche zu implementieren. Die wichtigste Methode von ServiceStateComponentView ist `paintYou(..)`. Sie wird jedes Mal aufgerufen, wenn

das Swing-System feststellt, dass der Zustand neu gezeichnet werden muss. Als Parameter bekommt sie ein Objekt der Swing-Klasse Graphics übergeben, die elementare Zeichenfunktionen anbietet, um damit das gewünschte Symbol für einen Service-Zustand zu zeichnen. Weiterhin sind der Klasse ServiceStateComponentView zwei Controller-Komponenten zugeordnet, *StateMouseListener* und *StateMouseMotion-Listener*. Sie reagieren auf alle Mausereignisse, die auftreten, wenn der Benutzer den Service-Zustand mit der Maus anwählt, verschiebt usw. Sie müssen intern herausfinden, welches Ereignis ausgelöst wurde und die entsprechenden Aktionen durchführen.

10 Ausführung von XML-Prozessen

Im vorherigen Kapitel wurde das im Rahmen dieser Arbeit entwickelte Modellierungswerkzeug vorgestellt, mit dem sich XML-Prozesse in Form von Prozessdiagrammen entwerfen lassen. In den folgenden Abschnitten soll nun betrachtet werden, wie sich die so definierten Prozesse ausführen lassen. Als Basis für die Ausführung dienen die Prozessbeschreibungen in der Sprache PML (vgl. Kapitel 8), die vom Modellierungswerkzeug erzeugt wurden. Zunächst erfolgt eine Betrachtung zur Semantik von Prozessdiagrammen, bevor dann der Interpreter vorgestellt wird, der die Ausführung übernimmt. Anschließend wird darauf eingegangen, was bei der Entwicklung von Konnektoren zu beachten ist, die die statischen Komponenten der Prozessdefinitionen darstellen und als Aktivitäten in den Prozessen verwendet werden können.

10.1 Semantik von Prozessdiagrammen

Bevor ein Interpreter entwickelt werden kann, der Prozessdiagramme ausführt, muss man sich Gedanken darüber machen, welche Semantik solche Diagramme überhaupt besitzen. Daher soll in diesem Abschnitt eine Semantik festgelegt werden. Die Aufgabe des Prozessinterpreters besteht dann darin, ein vorgegebenes Diagramm dementsprechend auszuführen. Die Beschreibung erfolgt zunächst informell anhand der in Abschnitt 7.3 definierten Stereotypen. Anschließend geben wir einen Algorithmus in Pseudocode-Notation an, der die Ausführung gemäß dieser Semantik leistet. Diese semi-formale Form der Semantikdefinition reicht aus, um die Funktionsweise des Prozessinterpreters so genau zu spezifizieren, dass eine Implementierung möglich wird.

Da UML-Aktivitätendiagramme in der UML-Spezifikation [Omg01] als Spezialfall von Zustandsmaschinen angesehen werden, leitet sich auch ihre Semantik aus der von Zustandsmaschinen ab. Die Besonderheit von Aktivitätendiagrammen sind dabei die sogenannten Aktionszustände. In [Omg01] (2-184) wird ihnen folgende Semantik zugeordnet: Sobald eine eingehende Transition eines Aktionszustands ausgelöst wird, beginnt die Ausführung der Aktion, die mit dem Zustand verknüpft ist. Wenn die Aktion beendet ist, werden die ausgehenden Transitionen des Zustands aktiviert. Da Pseudozustände - die zweite wichtige Zustandsart in Aktivitätendiagrammen - keine zugeordnete Aktion besitzen, werden ihre ausgehenden Transitionen sofort aktiviert, sobald eine eingehende bzw. im Falle eines Join-Zustands alle eingehenden getriggert werden. Beginnend beim Startzustand eines Diagramms wird dies nun iterativ durchgeführt, bis der Endzustand erreicht ist. In diesem Augenblick gilt die Ausführung des Diagramms als beendet.

Diese von der UML-Spezifikation vorgeschlagene Ausführungssemantik ist laut [Wie01] im Zusammenhang mit Workflow-Management-Systemen problematisch. Das liegt daran, dass die UML-Spezifikation davon ausgeht, dass mit einem Aktivitätendiagramm ein Softwaresystem spezifiziert wird, das selbst die Aktion der Zustände ausführt. Innerhalb eines sogenannten *run-to-completion-step* werden die Aktionen aller aktiven Zustände ausgeführt. Erst wenn dies beendet ist, beginnt der nächste run-to-completion-step. In Workflow-Management-Systemen liegen jedoch andere Voraussetzungen vor. Das WFMS ist lediglich für die korrekte Abhandlung des Kontrollflusses

zuständig, die Ausführung der Aktionen selbst delegiert es an die dafür zuständigen Akteure, in unserem Fall die Softwarekomponenten. Vereinfacht gesagt besteht die Aufgabe des WFMS in der Ausführung der Transitionen eines Diagramms, die der Akteure in der Ausführung der Aktionszustände. Um diesen Sachverhalt zu verdeutlichen, wird in [Wie01] folgendes Beispiel verwendet:

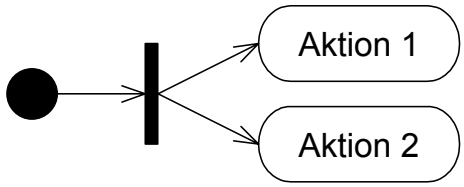


Abb. 10.1: Beispiel zur Semantik von Aktivitätendiagrammen

Nach der UML-Spezifikation werden die beiden Aktionen 1 und 2 simultan innerhalb desselben run-to-completion-step ausgeführt, d.h. ihre Ausführung ist zur gleichen Zeit beendet. Wenn eine solche Konstruktion in einem Workflow modelliert wird, ist jedoch meistens eine andere Semantik beabsichtigt, nämlich dass die Ausführung der Aktionen zur gleichen Zeit beginnen soll. Da die Aktionen selbst an Akteure delegiert werden, können sich die Zeiten, zu denen die Ausführungen enden, erheblich unterscheiden. Möglicherweise wurden im oberen Prozessstrang bereits mehrere Folgezustände bearbeitet, während sich der untere immer noch in der dargestellten Aktion 2 befindet. Aufgrund dieser Probleme wird in [Wie01] eine eigene Ausführungssemantik für Aktivitätendiagramme beschrieben, die sich für den Einsatz im Workflow-Management besser eignet, sowie ein Algorithmus angegeben, der die Diagramme entsprechend dieser Semantik ausführt. Im Mittelpunkt der Betrachtungen stehen dabei Ereignisse, die innerhalb des Systems auftreten und ihre Behandlung innerhalb der Workflow-Ausführung. Wie bereits im Abschnitt 6.2.6 argumentiert, haben wir für Prozessdiagramme auf den Einsatz von Ereignissen verzichtet. Die Verwendung des Algorithmus aus [Wie01] würde also für Prozessdiagramme einen großen Overhead bedeuten. Aus diesem Grund übernehmen wir zwar insoweit die dort eingeführte Semantik, wie es für Prozessdiagramme erforderlich ist, entwickeln aber einen eigenen Algorithmus, der speziell auf die Ausführung von Prozessdiagrammen zugeschnitten ist. Neben dem Fehlen von Ereignissen, sind in Prozessdiagrammen folgende Konstrukte von Aktivitätendiagrammen nicht erforderlich (vgl. Abschnitt 6.2.6):

1. Prozessdiagramme erlauben neben Pseudozuständen ausschließlich Aktions- und Subaktivitätszustände. Daher ergeben sich die folgenden beiden Ausführungsregeln:

   - Sobald die eingehende Transition eines Aktionszustands geschaltet hat, wird die zuständige Softwarekomponente mit der Ausführung beauftragt. Wenn sie terminiert, wird die ausgehende Transition des Zustands aktiviert.
   - Sobald die eingehende Transition eines Subaktivitätszustands geschaltet hat, wird der Ausführungsalgorithmus für den referenzierten oder eingebetteten Prozess gestartet. Wenn er terminiert, wird die ausgehende Transition des Zustands aktiviert.

2. Da mit dem XML-Migrationsdokument ein impliziter Datenfluss gegeben ist, ist eine explizite Modellierung mit Hilfe von Objektflusszuständen nicht vorgesehen. Damit entfällt für Prozessdiagramme auch die Semantik dieser Zustände.

3. Weil in XML-Prozessen keine Organisationshierarchie gegeben ist, sondern als Akteure ausschließlich Softwarekomponenten zum Einsatz kommen, werden zur Modellierung keine Bahnen (Swimlanes) verwendet.

Neben diesen Einschränkungen enthalten Prozessdiagramme Erweiterungen, für die wir im Folgenden die Semantik festlegen. Wie schon die Erweiterungen selbst (siehe Abschnitt 7.3) soll auch die Beschreibung der Semantik anhand der definierten UML-Stereotypen erfolgen.

Stereotyp «processDiagram»

Jedem Prozessdiagramm ist das Stereotyp «processDiagram» zugeordnet, so dass es die Attribute „inputPort“ und „outputPort“ besitzt, um Ein- und Ausgangsport zu spezifizieren. Nur wenn ein XML-Dokument konform zum Eingangsport ist, kann es entsprechend der definierten Semantik durch den Prozess verarbeitet werden. Bei erfolgreicher Ausführung des Prozesses entspricht es anschließend dessen Ausgangsport. Wurde zusätzlich das Attribut „transactional“ gesetzt, so wird der Prozess entweder vollständig oder gar nicht ausgeführt, nicht jedoch teilweise. Wenn bei seiner Ausführung ein Fehler auftritt, ist dafür zu sorgen, dass alle Effekte, die sich bis dahin ergeben haben, zurückgeführt werden, so dass sich das System wieder im Ausgangszustand befindet (vgl. auch Abschnitt 10.4).

Stereotyp «stateWithPorts»

Das Konzept der Ports, die in Abschnitt 6.3 eingeführt wurden, hat zunächst eine wichtige Bedeutung bei der Modellierung von Prozessdiagrammen, um Inkonsistenzen so früh wie möglich zu erkennen oder zu vermeiden. Sie bieten jedoch auch die Möglichkeit, bei der Ausführung eines Prozesses Konsistenzprüfungen durchzuführen. Jedem Zustand eines Prozessdiagramms sind über das Stereotyp «stateWithPorts» Ports zugeordnet. Bevor die Aktion, die mit einem Zustand verbunden ist, ausgeführt wird, wird überprüft, ob das Migrationsdokument zum Eingangsport des Zustands konform ist. Nach Ausführung der Aktion wird das Dokument gegen den Ausgangsport validiert. Falls bei der Prüfung der Ports Fehler auftreten, befindet sich der Prozess in einem inkonsistenten Zustand und muss abgebrochen werden.

Die meisten in Prozessdiagrammen erlaubten Zustandsarten haben dieselbe Semantik wie in UML-Aktivitätendiagrammen (siehe [Omg01] 2-175ff). Ausnahmen bilden Service-Zustände (siehe nächsten Absatz) sowie Join- und Synchjoin-Zustände. Letztere haben nicht nur die Aufgabe einer zeitlichen Synchronisation, sondern sie müssen auch die Migrationsdokumente aller ihrer eingehenden Transitionen sammeln und daraus gemäß der Vorschrift aus Abschnitt 6.2.5 ein einzelnes Dokument erzeugen. Dieses wird dann an den Rest des Prozesses weitergeleitet.

Stereotyp «serviceState»

Die wichtigste Zustandsart in Prozessdiagrammen sind Service-Zustände. Jedem Service-Zustand ist über das Metamodell für Prozessdiagramme eine Operation mit dem Stereotyp «service» zugeordnet. Wird bei der Ausführung des Prozesses ein solcher Zustand erreicht, wird das Migrationsdokument an die Softwarekomponente übergeben, zu der der Service gehört. Zuvor erfolgt eine Konformitätsprüfung anhand des Eingangsports. Es müssen dann die Argumentwerte berechnet werden, sofern der Service Parameter verlangt. Für jeden Parameter ist im Prozessdiagramm entweder ein statischer Wert oder ein XPath-Ausdruck angegeben, der sich auf das Migrationsdokument oder die Prozessumgebung bezieht (vgl. Abschnitte 6.2.2). Jedes Argument wird ausgewertet und sein Ergebnis beim Aufruf an den Service übergeben.

Nachdem der Service terminiert ist, wird überprüft, ob das Dokument bezüglich des Ausgangsports des Zustands gültig ist. Wenn dies nicht der Fall ist, muss davon ausgegangen werden, dass sich die Komponente nicht wie erwartet verhalten hat. In dieser Fehlersituationen ist die Konsistenz des Prozessdiagramms nicht gewahrt und seine Ausführung muss abgebrochen werden.

Stereotyp «typedTransition»

In Prozessdiagrammen bedeutet jede Transition neben dem Kontrollfluss auch einen Datenfluss, d.h. das Migrationsdokument wird an den oder die Folgezustände weitergereicht. Besitzt eine Transition einen Guard, so bezieht sich dieser entweder auf das Migrationsdokument oder die Prozessumgebung (vgl. Abschnitt 6.2.4). Bei der Ausführung des Prozesses muss der Guard jeder Transition ausgewertet werden. Nur wenn er wahr ist, kann die zugehörige Transition schalten. Weiterhin kann mit einer Transition eine Transformation des Migrationsdokuments verknüpft sein. Bei der Ausführung muss dafür gesorgt werden, dass diese durchgeführt wird, bevor das Dokument an den Zielzustand der Transition übergeben wird. Die auszuführende Transformation bestimmt dabei das XSL-Stylesheet des Transitionstyps, der der Transition zugeordnet ist. Es wird im Attribut „transformation“ des Stereotyps «transitionType» festgelegt.

Interpretation von Prozessdiagrammen

Nachdem nun eine Semantik für Prozessdiagramme festgelegt wurde, kann ein System entwickelt werden, dass ein XML-Eingabedokument und eine Prozessbeschreibung als Eingabe erhält und diese Schritt für Schritt abarbeitet. Aufgrund dieser Funktionsweise sprechen wir in dieser Arbeit von einem „Prozessinterpreter“.

Alternativ könnte für die Ausführung auch ein Übersetzer zum Einsatz kommen. Dieser würde die PML-Beschreibung eines Prozesses einlesen und daraus ausführbaren Code, also beispielsweise eine Java-Klasse, erzeugen, der dann direkt ausgeführt werden könnte, ohne dass ein Interpreter den Prozess schrittweise abarbeitet. Die Verwendung eines Übersetzers bedeutet vor allem für solche Prozesse Effizienzverbesserungen, die einmal definiert und dann häufig ausgeführt werden. Es ist jedoch auch denkbar, dass ein Benutzer einen individuellen Prozess modelliert und diesen gemeinsam mit dem Eingabedokument an die ausführende Instanz sendet. Ein solcher Prozess würde vermutlich nur einmal ausgeführt, so dass eine zuvor notwendige Übersetzung in eine Java-Klasse zusätzlichen und unnötigen Rechenaufwand bedeutete. Ein weiterer Nach-

teil der Übersetzung ist, dass bei jeder Änderung eines Prozessdiagramms der Code neu erzeugt werden muss. Im Falle des Interpreters sind Änderungen dagegen sofort wirksam, nachdem die neue PML-Beschreibung gespeichert ist. Wird haben uns daher entschlossen, zunächst Konzepte zur Interpretation von Prozessdiagrammen zu entwickeln. In weiterführenden Arbeiten könnte dann untersucht werden, wie sich durch den Einsatz eines Übersetzers sinnvoll Effizienzsteigerungen erzielen ließen (vgl. Abschnitt 12.2).

Der Interpreter arbeitet gemäß dem oben diskutierten, in [Wie01] beschriebenen Prinzip, wonach das WFMS für das Management der Prozessausführung und die Behandlung der Transitionen zuständig ist, es jedoch alle Aktionen innerhalb von Aktionszuständen an Softwarekomponenten delegiert. Zur Erfüllung dieser Aufgaben enthält der Interpreter eine Reihe verschiedener Bestandteile, die im Abschnitt 10.2 vorgestellt werden. Der folgende Algorithmus führt einen Prozess gemäß der obigen Semantikdefinition aus:

```
executeProcess(process, document):
1  executeSection(process.getInitialState(), document)

executeSection(state, document):
2  firstState := true
3  do
4     if firstState then
5           firstState := false
6     else
7           state := getFollowing(state)
8           transition := getUsedTransition(state)
            /* Transition über die der Folgezustand
               erreicht wurde */
9           stylesheet := transition.getStylesheet()
10          if stylesheet ≠ null then
11                transform(document, stylesheet)
12          end if
13    end if
14    validateAgainstPort(document, state.getInputPort())
15    document := execute(state)
16    validateAgainstPort(document, state.getOutputPort())
17 while (not (state is JoinState or state is FinalState))
18 if this is parallel subsection then
19    notifyParentSection(document)
20 end if

getFollowing(state):
21 if state is DecisionState then
22    states := state.getSuccessors()
23    for i:=1 to states.length do
24          following := states[i]
25          transition := getUsedTransition(following)
            /* Transition über die der Folgezustand
               erreicht wurde */
26          guard := transition.getGuard()
27          if guard.isElse() then
28                elseState := following
29          else if guard.isTrue() then
30                return following
31          end if
32    end for
33    return elseState
```

```
34 else if state is ForkState then
35    return state.getCorrespondingJoinState()
36 ...
37 else /* Zustand mit genau einer ausgehenden Transition */
38    return state.getSuccessor()
39 end if

execute(state):
40 if state is ForkState then
41    states := state.getSuccessors()
42    for i:=1 to states.length do parallel
43          successor := states[i]
44          transition := getUsedTransition(successor)
            /* Transition über die der Folgezustand
               erreicht wurde */
45          if transition.getGuard().isTrue() then
46                executeSection(successor, document.clone())
47          end if
48    end for
49    documents := waitForSubSections()
50    document := createJoinedDocument(documents)
51 else if state is ServiceState then
52    document := state.getService().invoke(document)
53 else ...
54 end if
55 return document
```

Abb. 10.2: Algorithmus für die Ausführung eines Prozesses

Aus Platzgründen zeigt die Abbildung nur auszugsweise die wichtigsten Bestandteile des Ausführungsalgorithmus. Die zentrale Methode ist `executeSection(..)`, die sowohl dazu verwendet wird, einen gesamten Prozess, als auch parallele Stränge im Zusammenhang mit Fork-Zuständen auszuführen. Sie arbeitet Schritt für Schritt den Prozess ab und veranlasst die notwendigen Aktivitäten wie Transformation des Migrationsdokuments (Zeile 11), Validierung gegen Ports (Zeile 14 und 16) usw.

Für jeden Zustand wird mit Hilfe der Methode `getFollowing(..)` der Folgezustand bestimmt (Zeile 7). Die Abbildung zeigt die dazu notwendigen Berechnungen für Decision-, Fork- und solche Zustände, die genau einen Nachfolger haben. Bei Fork-Zuständen wird hier der zugehörige Join-Zustand zurückgegeben, in dem die parallelen Teilstränge wieder zusammengeführt werden. Ihre Ausführung geschieht in der Methode `execute(..)`, indem für jeden Strang rekursiv die Methode `executeSection(..)` aufgerufen wird. Sind alle Stränge gestartet, wartet die Ausführung auf ihre Terminierung (Zeile 49). Jeder Strang meldet sich, sobald er beendet ist und übergibt dabei zugleich sein Migrationsdokument (Zeile 19). Erst wenn alle Stränge bis zum Ende abgearbeitet wurden und ihr Dokument abgelegt haben, wird die Ausführung fortgesetzt und ein gemeinsames Ergebnisdokument konstruiert (Zeile 50). Die Ausführung des gesamten Prozesses ist beendet, sobald der oberste Aufruf der Methode `executeSection(..)` terminiert, dem der Startzustand des Prozesses als Parameter übergeben wurde.

Nachdem nun die Semantik von Prozessdiagrammen festgelegt und darauf aufbauend ein Ausführungsalgorithmus für Prozesse entworfen wurde, haben wir den Prozessinterpreter entwickelt, der im folgenden Abschnitt beschrieben wird.

10.2 Der Prozessinterpreter

Die Aufgabe des Prozessinterpreters besteht darin, die mit dem Modellierungswerkzeug entworfenen und in PML gespeicherten Prozessmodelle entsprechend der im letzten Abschnitt festgelegten Semantik auszuführen. Er besitzt dazu eine Reihe von Klassen für unterschiedliche Bereiche wie interne Repräsentation der Prozessmodelle, Aufruf von Konnektoren, Synchronisation und andere. Wie schon das Modellierungswerkzeug wurde auch der Interpreter in der Sprache Java realisiert. In den folgenden Abschnitten werden die jeweils wichtigsten Bestandteile erläutert. Alle Klassen des Interpreters bestehen zusammen aus ungefähr 5.000 LOC (lines of code).

10.2.1 Interne Datenstruktur zum Zugriff auf Prozessmodelle

Es gibt prinzipiell zwei verschiedene Möglichkeiten für die Verwaltung der Prozessmodelle durch den Interpreter. Zum einen könnte er geeignete XML-Werkzeuge verwenden, um zu einem Prozess, der ausgeführt werden soll, die PML-Beschreibung einzulesen. Er hätte anschließend beispielsweise einen DOM-Baum vorliegen und könnte direkt anhand dieser Datenstruktur den Prozess schrittweise abarbeiten und die notwendigen Daten aus dem DOM auslesen. Da jedoch der Zugriff auf XML-Daten insbesondere mit Hilfe von XPath-Operationen vergleichsweise viel Rechenzeit in Anspruch nimmt und zudem ein DOM-Baum hohen Speicherbedarf besitzt, ist diese Variante zu ineffizient.

Wir haben uns daher entschieden, die PML-Dateien zunächst mit Hilfe eines Parsers einzulesen und anschließend in das implementierte Metamodell für Prozessdiagramme (vgl. Abschnitt 7.4) zu überführen. Diese Objektstruktur erlaubt effizientere Zugriffe auf die Prozessmodelle und kann zum Beispiel die Nachfolgezustände eines Zustands durch direktes Auswerten des zugehörigen Objektattributs bestimmen. Auch für das Modellierungswerkzeug wurde schon diese Datenstruktur verwendet (vgl. Kapitel 9). Der Vorteil dieses Vorgehens kommt besonders dann zum Tragen, wenn der Interpreter nicht bei jeder Prozessausführung die PML-Beschreibung einliest und transformiert, sondern einen Pufferungs-Mechanismus (Caching, vgl. Abschnitt 12.2) verwendet, der die internen Prozessbeschreibungen im Speicher hält, so dass sie bei der Abarbeitung anderer Prozessinstanzen wiederverwendet werden können. Die Rechenzeit für die Umwandlung des PML-DOM in die interne Objektstruktur ist dann nur einmal zu investieren. Das Einlesen und Umwandeln von PML-Dateien übernimmt die aus Abschnitt 8.4.1 bekannte Klasse PMLImporter.

Schnittstellen zu den Klassen des Metamodells

Um einen Prozess anhand der internen Datenklassen ausführen zu können, benötigt der Interpreter zusätzliche Funktionalität von diesen Klassen, die in ihnen zunächst nicht enthalten ist. Zu diesem Zweck verwenden wir das Entwurfsmuster *Adapter*, mit dem sich die Schnittstelle der Datenklassen anpassen lässt (vgl. [Gam96] S.171). Abbildung 10.3 verdeutlicht den Einsatz des Musters.

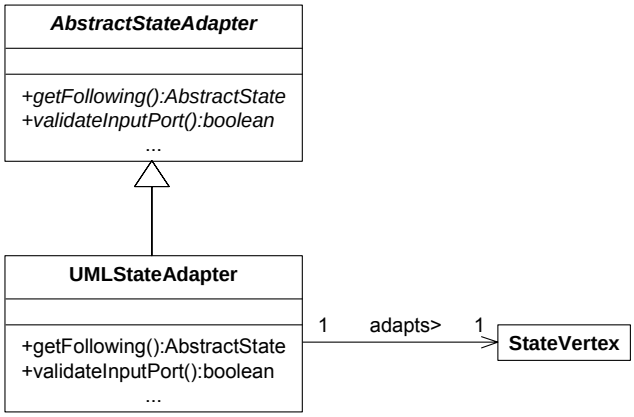


Abb. 10.3: Adapterklassen

Durch die Verwendung der abstrakten Klasse *AbstractStateAdapter* wurde die Möglichkeit offengelassen, später ein anderes Datenmodell zu Grunde zu legen, ggf. doch den PML-DOM-Baum, falls dies sinnvoller erscheint. Es muss dann lediglich die Klasse UMLStateAdapter, die die Anpassung für unsere Datenstruktur realisiert, durch eine andere Implementierung ersetzt werden. Der Rest des Interpreters bleibt dabei unverändert. Leider konnte zur Anpassung der Datenstruktur in diesem Fall nicht die aus Abschnitt 7.4 bekannte Klasse AbstractModelElementAdapter verwendet werden, da UMLStateAdapter bereits eine Oberklasse besitzt und nicht zusätzlich davon erben könnte. Die Klasse hat daher eine eigene Referenz auf StateVertex.

Der Prozessinterpreter kann jedoch die Klasse AbstractStateAdapter nicht direkt als Repräsentation eines Prozesszustandes verwenden. Es fehlt eine Methode `execute()`, die die Aktionen ausführt, die zu einem bestimmten Prozesszustand gehören. Diese Aktionen sind in hohem Maße abhängig vom Typ des Zustandes, d.h. ob es sich um einen Service-, Fork- oder anderen Zustand handelt. Aus diesem Grund wird zusätzlich das Entwurfsmuster *Bridge* verwendet. Dieses Muster dient dazu, eine Abstraktion (hier: AbstractState) von seiner Implementierung (hier: AbstractStateAdapter) zu trennen und beide unabhängig voneinander vererben zu können (vgl. [Gam96] S.186). Auf diese Weise können sowohl Unterklassen für die verschiedenen Zustandstypen gebildet werden, als auch die Möglichkeit beibehalten werden, wie oben beschrieben das zu Grunde liegende Datenmodell zu verändern. Abbildung 10.4 zeigt die notwendigen Klassen. *AbstractState* bietet so die passende Schnittstelle, die der Prozessinterpreter zu einem Zustand benötigt.

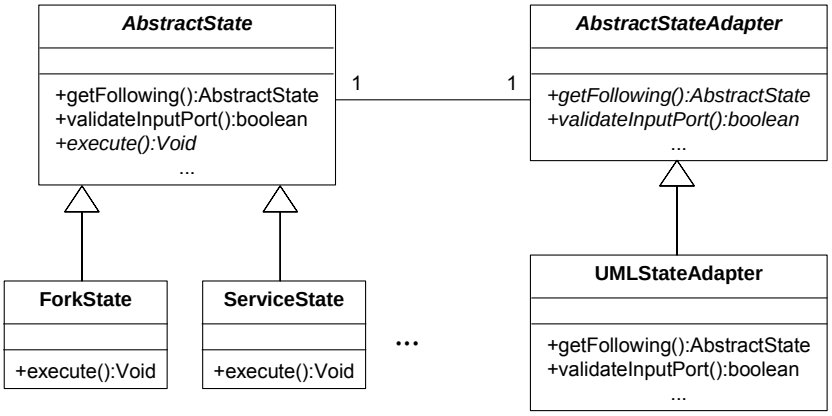


Abb. 10.4: Verwendung des Entwurfsmusters Bridge

Abbildung 10.5 zeigt zur Verdeutlichung das Zusammenwirken der beiden Muster am Beispiel eines Service-Zustandes und die sich daraus ergebende Schichtenstruktur.

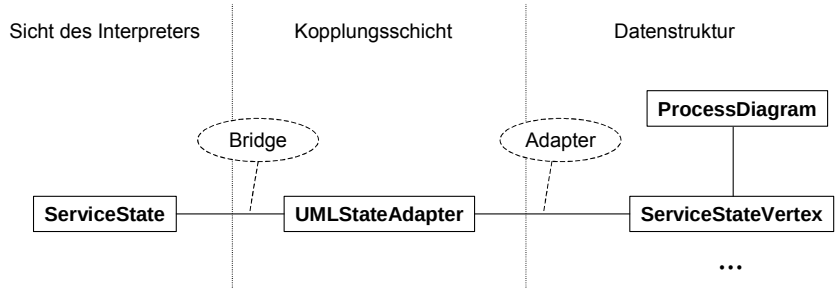


Abb. 10.5: Verwendung der Datenklassen durch den Prozessinterpreter

10.2.2 Ausführende Klassen

Mit Hilfe der im vorherigen Abschnitt beschriebenen Klasse AbstractState kann der Prozessinterpreter den Prozess schrittweise abarbeiten. Wie bereits im Abschnitt 10.1 erläutert, ist der zentrale Ausführungsalgorithmus in der Lage, einen Prozessabschnitt auszuführen. Ein solcher Abschnitt kann dabei entweder der gesamte Prozess, ein referenzierter oder eingebetteter Subprozess oder ein paralleler Teilstrang im Zusammenhang mit einem Fork-Zustand sein. Bei parallelen Teilsträngen müssen mehrere Threads gestartet werden, von denen jeder einen Teilprozess abarbeitet. Dies lässt sich sehr leicht realisieren, indem eine Unterklasse der Java-Klasse *Thread* gebildet wird. Die zentrale Klasse des Interpreters ist daher *ProcessSection*, die die notwendige Funktionalität besitzt, um einen Prozessabschnitt auszuführen. Davon abgeleitet ist die Klasse *ProcessInterpreter*, die aus der Systemumgebung heraus instanziiert werden kann, um ein gesamtes Prozessmodell auszuführen. Abbildung 10.6 zeigt die Zusammenhänge.

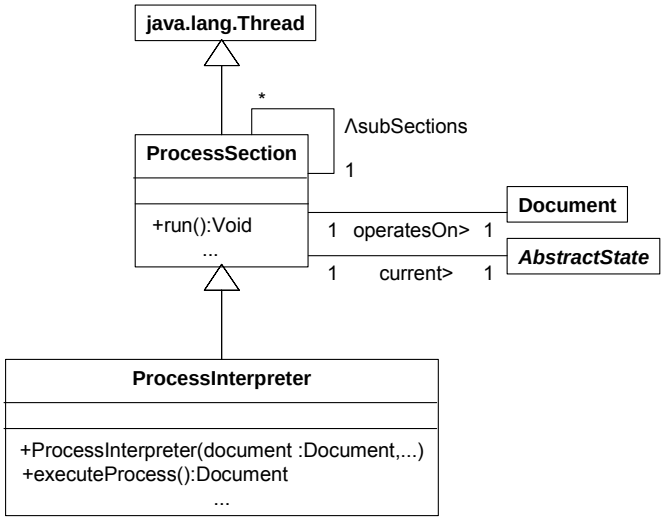


Abb. 10.6: Ausführende Klassen

Die Klasse ProcessSection besitzt zunächst Assoziationen zu dem Migrationsdokument sowie dem aktuellen Zustand. In der Methode `run()`, die aus der Oberklasse Thread überschrieben wurde, befindet sich der aus Abschnitt 10.1 bekannte Algorithmus, der einen Prozessabschnitt ausführt. Wenn dabei ein Forkzustand erreicht wird, erzeugt der Interpreter für jeden parallelen Strang eine neue Instanz der Klasse ProcessSection, die als Kind an die übergeordnete ProcessSection angehängt wird (subSections). Diese wartet nun solange, bis alle ihre Unterabschnitte terminiert sind. Anschließend sammelt sie die Ergebnisdokumente der Unterabschnitte ein und erzeugt daraus ein neues Migrationdokument (vgl. Abschnitt 6.2.5). Danach setzt sie die Ausführung ihres Prozessabschnitts fort. Auf die gleiche Weise geht der Interpreter im Falle eines Subaktivitätszustandes vor. Auch hier wird für den eingebetteten Prozess eine eigene ProcessSection gestartet. Durch diesen Mechanismus ist es auf einfache Weise möglich, beliebig tief geschachtelte Prozesse korrekt abzuarbeiten.

Die Methode `executeProcess()` aus der Klasse ProcessInterpreter wird von der Systemumgebung aus verwendet, um ein Prozessmodell auszuführen. Sie startet und initialisiert den obersten Thread in der Hierachie der Prozessabschnitte. Wenn dieser terminiert, ist die Abarbeitung des gesamten Prozesses beendet und die Methode liefert als Ergebnis das resultierende Migrationsdokument. Abbildung 10.7 zeigt ein vereinfachtes Prozessdiagramm und die zugehörige Objektstruktur zur Ausführungszeit. Auf der obersten Ebene befindet sich der ProcessInterpreter, der aus der Umgebung heraus gestartet wurde, um den Prozess auszuführen. Für jeden der beiden parallelen Stränge wird eine eigene ProcessSection als Kind an ProcessInterpreter angehängt. Dieser fährt erst dann fort, wenn beide Kinder terminiert sind. Da der linke Strang aus einem Subprozess besteht, wird für diesen ebenfalls eine ProcessSection als Kind des linken Strangs erzeugt.

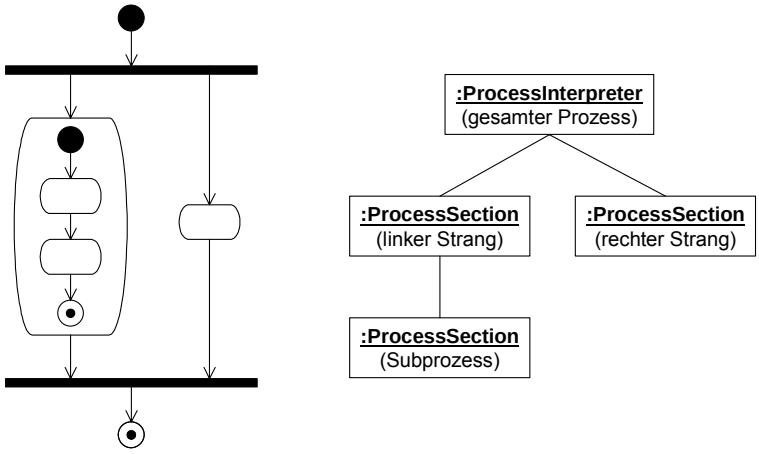


Abb. 10.7: Schachtelung von Prozessabschnitten zur Laufzeit

Abbildung 10.8 zeigt ein grobes Aktivitätendiagramm des Algorithmus in der Methode `run()` aus der Klasse ProcessSection, in dem aus Platzgründen nicht alle Feinheiten aufgeführt sind. Es zeigt die Abläufe innerhalb der Methode, die zur Ausführung eines Prozessabschnitts bzw. eines gesamten Prozesses dienen (vgl. Abschnitt 10.1).

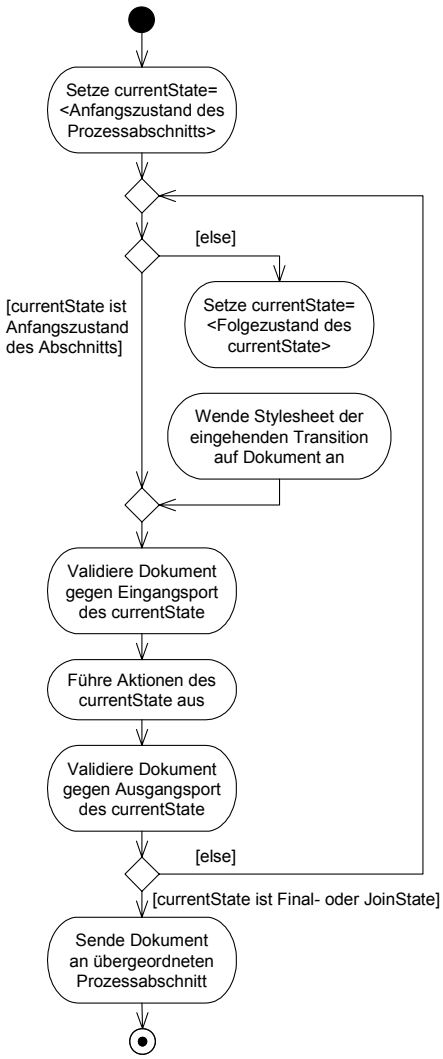


Abb. 10.8: Ausführung eines Prozessabschnitts

Um das Migrationsdokument gegen einen Port zu validieren, generiert der Interpreter einen XPath-Ausdruck. Dieser überprüft, ob das Wurzelelement des Dokuments zu mindestens einem Dokumenttypen des Ports konform ist (vgl. Abschnitt 6.3). Ist dies der Fall, ist das Dokument typkonsistent, andernfalls ist ein Fehler aufgetreten und die Ausführung des Prozesses wird abgebrochen.

Alle Aktionen wie Aufruf eines Service oder Erzeugen der ProcessSections für parallele Stränge geschieht bei der Ausführung des aktuellen Zustands (currentState). Wenn der Folgezustand des aktuellen Zustands bestimmt wird, müssen ggf. Guard-Ausdrücke ausgewertet werden, um zu prüfen, welchen Ausführungspfad der Prozess nehmen soll. Dazu existiert die Hilfsklasse *GuardEvaluator*. Ein Guard muss immer folgenden Aufbau haben, damit er vom Interpreter ausgewertet werden kann: <Basis>:<XPath>. Dabei ist <XPath> durch einen gültigen XPath-Ausdruck zu ersetzen. <Basis> muss entweder das Wort „document“ sein, um zu signalisieren, dass sich der XPath-Ausdruck auf das Migrationdokument beziehen soll, oder ein anderer Name, der für ein XML-Dokument innerhalb der Prozessumgebung steht (vgl. Abschnitt 10.5), auf das der XPath-Ausdruck angewendet wird. Erlaubte Ausdrücke sind also beispielsweise:

```
document:/root/element1='wert'
env1:local-name(/root/*)='element1'
```

10.2.3 Synchronisationsmechanismus

Prozessdiagramme erlauben es, mit Hilfe von Synchronisationszuständen eine Synchronisation zwischen unterschiedlichen parallelen Teilprozessen zu realisieren. Im Abschnitt 6.2.5 wurde beschrieben, dass dabei nicht nur eine zeitliche Abstimmung, sondern auch ein Austausch des Migrationsdokuments stattfindet. Für die Umsetzung im Prozessinterpreter wurde auf das Konzept der sogenannten *Monitore* zurückgegriffen. Laut [Bac98] (S.311ff) ist ein Monitor eine Klasse, die bestimmte Daten kapselt. Alle Operationen auf diesen Daten werden unter gegenseitigem Ausschluss, d.h. als kritischer Abschnitt ausgeführt. Zusätzlich findet eine Bedingungssynchronisation statt, damit sich die Daten stets in einem gültigen Zustand befinden. Eine Operation muss ggf. solange warten, bis die Daten sich in einem Zustand befinden, der garantieren kann, dass die Daten nach Ausführung der Operation ebenfalls konsistent sind. Die Klasse *SynchMonitor* wurde nach diesem Prinzip realisiert.

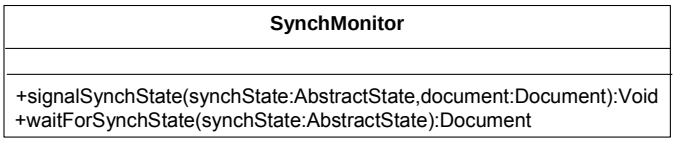


Abb. 10.9: Synchronisationsklasse SynchMonitor

Durch Aufruf der Methode `signalSynchState(..)` signalisiert ein Prozess seine Ankunft an einem bestimmten Synchronisationszustand. Er muss dabei neben einer

Referenz auf diesen Zustand auch eine Kopie seines aktuellen Migrationsdokuments übergeben, da dieses vom Synchronisationspartner benötigt wird. Dieser ruft die Methode `waitForSynchState(..)` auf, wenn er an dem Zustand ankommt. Falls der Partner noch nicht signalisiert hat, muss der Prozess solange warten. Andernfalls bekommt er das Dokument des Partners zurückgeliefert, das dieser im Monitor abgelegt hat. Es wird dann gemäß Abschnitt 6.2.5 ein neues Dokument zusammengesetzt, das der Prozess anschließend verarbeiten kann.

10.2.4 Aufruf von Services

Im Abschnitt 6.1.3 wurden die Vorteile aufgeführt, die sich aus einer dynamischen Komponentenbindung ergeben. Dabei sollen in den statischen Modellen grundsätzlich nur Interfaces angegeben werden, die in den Prozessdiagrammen mit den Service-Zuständen verknüpft werden. Erst zur Laufzeit soll der Prozessinterpreter eine konkrete Implementierung finden, die das Interface erfüllt und die er für die Ausführung des Service verwendet. Um zu diesem Zweck verschiedene Mechanismen zu erlauben, existiert das Interface *IConnectorFactory*.

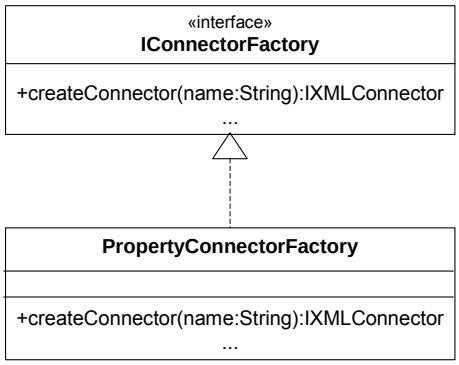


Abb. 10.10: Verwaltung von Konnektorinstanzen

Der Interpreter ruft die Methode `createConnector(..)` auf, wenn er für die Ausführung eines Service eine Instanz des zugehörigen Konnektors benötigt. Als einfachste Möglichkeit zur Erzeugung einer solchen Instanz wurde von uns die Klasse *PropertyConnectorFactory* realisiert. Sie verwendet eine XML-Konfigurationsdatei, in der jedem Interface eine passende Klasse zugeordnet ist. Um die Effizienz der Verwaltung von Konnektorinstanzen zu verbessern, könnte eine Art Pool angelegt werden. Wenn der Interpreter einen Konnektor benötigt, würde dann zunächst im Pool nachgesehen, ob sich bereits eine passende Instanz darin befindet, die wiederverwendet werden kann. Nur wenn dies nicht der Fall ist, wird eine neue Instanz erzeugt. Zur Erläuterung des Interface IXMLConnector siehe Abschnitt 10.5.

Als Alternative zu einer solchen festen Zuordnung von Klassen zu Konnektor-Interfaces, könnten auch intelligente Suchmechanismen zum Einsatz kommen. Einen interessanten Ansatz in dieser Richtung stellt die neue Java-Technologie *Jini* dar (siehe [Jin01]). Sie stellt eine Infrastruktur zur Verwaltung dynamischer Dienste in Netzwerken zur Verfügung. Die Dienste können sich beim System an- und abmelden; Komponenten, die einen bestimmten Dienst in Anspruch nehmen möchten, können diesen im Netzwerk anhand eines Interfaces suchen, den der Dienst erfüllen soll. Wir haben uns

aus Zeitgründen auf die einfache Variante der Klasse PropertyConnectorFactory beschränkt. Der Interpreter ist jedoch so ausgelegt, dass sich jederzeit ohne gravierende Änderungen eine andere Lösung integrieren lässt.

Zum Aufruf eines Service dient die Hilfklasse *ServiceCall*. Im Prozessmodell wird für jeden Service-Zustand der Name eines Konnektor-Interfaces sowie einer seiner Service-Methoden angeben. Mit Hilfe der oben beschriebenen ConnectorFactory erlangt die Klasse ServiceCall die Instanz eines Konnektors, der das geforderte Interface erfüllt. Anschließend lässt sich in Java dynamisch ein Methodenaufruf der angegebenen Methode generieren. Dazu müssen Werte für alle Parameter der Methode übergeben werden. Zu diesem Zweck wurden in der Prozessbeschreibung Ausdrücke für jeden Parameter des Service angegeben. Diese müssen folgenden Aufbau haben, damit sie vom Interpreter ausgewertet werden können: <Basis>:<Ausdruck>. Für <Basis> sind drei Fälle zu unterscheiden, die denen im Abschnitt 6.2.2 entsprechen. Falls für <Basis> das Wort „static“ eingesetzt wird, so übergibt der Interpreter den Text für <Ausdruck> direkt als Argumentwert an den Service. Auf diese Weise lassen sich feste Werte in der Prozessbeschreibung verdrahten. Wenn für <Basis> das Wort „document“ angegeben wird, interpretiert der Interpreter <XPath> als XPath-Ausdruck, der auf das Migrationsdokument angewendet wird. In allen anderen Fällen geht er davon aus, dass sich in der Prozessumgebung (vgl. Abschnitt 10.5) ein XML-Dokument mit dem angegebenen Namen befindet, für den der XPath-Ausdruck ausgewertet werden soll. Gültige Argumentausdrücke sind also beispielsweise:

```
static:wert
document:/root/element1
env1:local-name(/root/*)
```

10.3 Fehlerbehandlung

Für die Robustheit des Prozessinterpreters und des Workflow-Management-Systems ist es sehr wichtig, dass auch unvorhergesehene Fehlersituationen eingeplant werden, die gerade bei der Integration von heterogenen Anwendungen auftreten können. Denn neben den Programmfehlern des Interpreters selbst, muss auch der Umgang mit Fehlern geklärt werden, die bei der Ausführung der zu integrierenden Komponenten und Anwendungen auftreten oder die sich durch die Kopplung und Kommunikation des Interpreters mit diesen Komponenten ergeben.

Die im Rahmen der Diplomarbeit durchgeführte, Java-basierte Implementierung des Prozessinterpreters benutzt zur grundlegenden Fehlerbehandlung das bekannte, in Java integrierte Verfahren der Ausnahmebehandlung (Exception-Handling). Die folgende Tabelle gibt einen Überblick über die wichtigsten Ausnahmetypen, die im Prozessinterpreter vorkommen können.

InvalidPMLDocumentException	Ein Fehler ist beim Laden der Prozessbeschreibung aus einem PML-Dokument aufgetreten; vermutlich aufgrund von Unstimmigkeiten oder inkonsistenten Daten im Dokument.
InvalidDocumentStructureExc.	Das Migrationsdokument entspricht nicht den Typvorgaben eines Ports.
GuardEvaluationException	Die Auswertung eines Guards ist fehlgeschlagen, weil der Ausdruck nicht gültig oder auf das Migrationsdokument nicht anwendbar war.
DocumentTransformationExc.	Die Transformation eines Dokuments durch einen Stylesheet-Prozessor schlug fehl oder das gesuchte Stylesheet konnte nicht gefunden werden.
CreateConnectorException	Beim Suchen bzw. Erzeugen einer Konnektorinstanz ist ein Fehler aufgetreten, möglicherweise konnte die Klasse nicht gefunden werden.
CallServiceException	Beim Aufruf eines Service ist ein Fehler aufgetreten, entweder weil die entsprechende Methode nicht gefunden wurde oder weil intern im Service ein Fehler aufgetreten ist, der nicht abgefangen werden konnte.
ProcessExecutionException	Ausnahme, die die anderen Ausnahmen kapselt und nach außen an die übergeordnete Instanz weitergibt.

Tab. 10.1: Typen von Ausnahmen beim Prozessinterpreter

Wenn aufgrund eines Fehlers eine dieser Ausnahmen auftritt, kann die Prozessausführung in der Regel nicht ohne Weiteres zu Ende geführt werden. Die Abarbeitung wird dann zwangsläufig gestoppt und der Fehler an die aufrufende Instanz gemeldet. Nun kann es aber vorkommen, dass Fehler bei einem Service auftreten, die nicht unbedingt den Abbruch des gesamten Prozesses bedingen müssen. Vielleicht gibt es zum Beispiel alternative Prozesszweige, die statt des fehlerhaften Teils ausgeführt werden könnten. Die folgenden Abschnitte enthalten Betrachtungen über Möglichkeiten, solche Fehler zu behandeln, und diskutieren die Zweckmäßigkeit, entsprechende Vorkehrungen im Prozessinterpreter zu treffen.

10.3.1 Fehlertoleranz

Die Grundidee fehlertoleranter Systeme ist, dass sie selbst dann korrekte Ergebnisse liefern, wenn während der Abarbeitung ein Fehler aufgetreten ist. Im Zusammenhang mit Prozessdiagrammen wäre eine einfache Möglichkeit zur Herstellung von Fehlertoleranz, beim Auftreten eines Fehlers in einem Service nicht den ganzen Prozess abzubrechen, sondern nur den Parallelstrang, in dem dieser Service aufgerufen wurde. Um dennoch eine vollständige Abarbeitung des Prozesses zu erreichen, müsste die Ausführung des abgebrochenen Prozessstrangs wiederholt werden. Dabei kann der Fehler zwar erneut auftreten, was aber nicht zwingend der Fall sein muss, vor allem dann nicht, wenn er beim ersten Mal nur deshalb aufgetreten ist, weil eine zeitlichen Schwankungen unterworfene Bedingung geherrscht hat. Als Beispiel könnte hier die zeitlich wechselnde Auslastung bzw. Überlastung eines Kommunikationsnetzwerkes genannt werden, die etwa die Arbeit eines verteilten Systems behindern kann. Handelt es sich um eine solche Art von Fehlern, könnte mit dem Wiederholungsversuch der Prozess doch noch korrekt abgearbeitet werden.

Zur Umsetzung des Konzepts müsste zu Beginn jedes parallelen Teilstrangs eine Sicherungskopie vom aktuellen Migrationsdokument angelegt werden, um für die Wiederholung gleiche Voraussetzungen hinsichtlich der Daten bereitzustellen. Es gibt bei der Wiederholung eines Teilstrangs allerdings zusätzlich das Problem, dass die vor der fehlerhaften Komponente bereits ausgeführten Services Seiteneffekte und Änderungen an externen Datenquellen verursacht haben könnten, die nicht ohne Weiteres durch eine Sicherungskopie rückgängig gemacht werden können. Werden diese Services beim Wiederholungsversuch ebenfalls ein zweites Mal mit ihren Nebeneffekten ausgeführt, kann das unter Umständen zu Konsistenzverletzungen führen.

Eine Verbesserung des Verfahrens hinsichtlich dieser Problematik bestände darin, nicht einen ganzen Teilstrang zurückzusetzen, sondern nur den einzelnen Service erneut aufzurufen, bei dem die Ausführung versagt hat. Das Problem der Seiteneffekte ist dann zwar kleiner, aber noch nicht gänzlich gelöst, weil auch innerhalb der Serviceausführung bis zu der Stelle, an der der Fehler aufgetreten ist, Nebeneffekte verursacht worden sein können, die nicht rückgängig zu machen sind. Außerdem sind jetzt viel öfter Sicherungskopien anzulegen, was zu Lasten der Performanz geht.

Unter Berücksichtigung der Tatsache, dass nicht für alle Services gleich hohe Erfolgsaussichten für die Umgehung eines Fehlers durch dieses Verfahren gegeben sind, erscheint es sehr zweifelhaft, ob diese Methode vom Prozessinterpreter standardmäßig bei allen Service-Fehlern angewandt werden sollte. Der Sinn der Methode hängt vielmehr von den Fehlertypen der einzelnen Services ab, je nach dem, ob sie durch einen Wiederholungsversuch behoben werden könnten oder nicht. Da der Interpreter diese Informationen über einen Service nicht kennt, erscheint es uns am zweckmäßigsten, solche Maßnahmen zur Fehlertoleranz vom Interpreter auf die einzelnen Konnektoren selbst zu verlagern. Der Entwickler, der die Interna des Service genau kennt, kann einen Toleranzmechanismus mit Wiederholungsversuch in den Konnektor einbauen, wenn ihm dies sinnvoll erscheint. Dadurch werden die Vorzüge der Fehlertoleranz-Idee gewahrt, ohne unnötige Performanzverluste dulden zu müssen.

10.3.2 Explizite Modellierung von Fehlerbehandlungen

Wie die vorausgegangenen Betrachtungen gezeigt haben, wird sich ein großer Teil möglicher Fehler nicht tolerieren lassen, sondern muss vom Prozessinterpreter behandelt oder weitergeleitet werden. Da die hier ausgeführten Workflow-Modelle auf der Verarbeitung von XML-Dokumenten durch Services basieren, wäre in Analogie zu den Ausnahmen in Java die Definition von eigenen XML-Elementen denkbar, die Informationen und Meldungen über aufgetretene Fehler speichern können. Wenn bei der Ausführung eines Service ein Fehler auftritt, könnte der Service dann anstelle des eigentlichen Migrationsdokuments ein XML-Dokument mit den Fehlerinformationen ausgeben, das dann ohne unkontrollierten Abbruch des Prozesses in einer explizit modellierten Fehlerbehandlung weiterverarbeitet werden könnte (vgl. Abbildung 10.11).

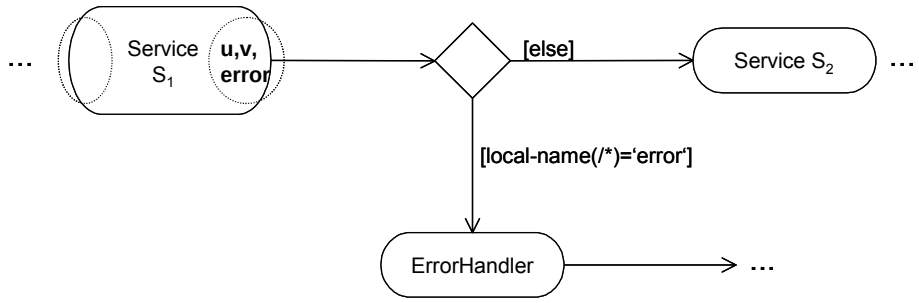


Abb. 10.11: Explizite Fehlerbehandlung im Prozessmodell

Man könnte überlegen, bei allen Service-internen Fehlern diese Methode anzuwenden und automatisiert eine Ausnahme (Exception) abzufangen, um sie in ein XML-Element umzuwandeln, das zum Beispiel unter dem Namen <error> vordefiniert wurde. Weil man Fehler nie ganz ausschließen kann, hätte dies allerdings zur Folge, dass unter Umständen nach jeder Aktivität im Prozessdiagramm ein XML-Dokument mit diesem Fehlerelement vorliegen kann und von den nachfolgenden Aktivitäten weiterverarbeitet werden muss. Als Folge enthielten alle Ports des Diagramms dieses Fehlerelement, so dass man es aufgrund des strengen Port-Konzepts bei allen Transitionstypen, Stylesheets etc. berücksichtigen müsste, auch wenn gerade der Einbau einer speziellen Fehlerbehandlung nicht sinnvoll wäre.

Darum haben wir für unsere Implementierung kein fixes Fehlerelement vordefiniert, das alle Ausnahmen aufnehmen kann, sondern überlassen es dem jeweiligen Modellierer und Entwickler der Konnektoren, Java-Ausnahmen durch spezielle, individuell definierte XML-Elemente im Migrationsdokument zu repräsentieren, die im weiteren Prozessverlauf zu einer besonderen Fehlerbehandlung führen.

10.4 Transaktionale Prozessausführung

Nachdem im letzten Abschnitt Überlegungen zur Behandlung von Fehlern während der Ausführung eines Prozessdiagramms gemacht wurden, soll hier speziell auf Transaktionen eingegangen werden. Wie in Abschnitt 7.3.2 erwähnt wurde, lässt sich ein Prozess oder Subprozess als *transaktional* definieren. Im Folgenden soll die Semantik dieser Festlegung betrachtet werden. Da sich das Thema Transaktionen als sehr umfangreich und komplex darstellt, haben wir aus Zeitgründen darauf verzichtet, entsprechende Konzepte im Prozessinterpreter umzusetzen. Ein Prozessdiagramm lässt sich also zwar als transaktional festlegen, dies hat aber keine Auswirkung auf die Ausführung mit Hilfe des von uns entwickelten Interpreters. In den folgenden Abschnitten sollen jedoch Vorschläge gemacht werden, wie sich dies prinzipiell integrieren ließe. Vergleiche dazu auch Abschnitt 2.2.2 über transaktionale Workflows.

10.4.1 Eigenschaften von Transaktionen

Eine Transaktion ist laut [Bac98] (S.483) eine Folge von Operationen auf einer Menge von Daten mit den sogenannten ACID-Eigenschaften:

Atomicity: Entweder werden alle Operationen einer Transaktion ausgeführt oder gar keine, d.h. die Transaktion wird entweder ganz oder gar nicht abgearbeitet, nicht jedoch teilweise. Falls während der Ausführung ein Fehler oder Systemausfall auftritt, müssen alle bis dahin vorgenommenen Änderungen an den Daten rückgängig gemacht werden.

Consistency: Jede Transaktion überführt das System aus einem konsistenten Zustand wiederum einen konsistenten Zustand, d.h. eine Transaktion darf die Daten nicht in einem ungültigen Zustand hinterlassen.

Isolation: Alle Änderungen, die die Operationen der Transaktion vornehmen, dürfen erst dann für den Rest des Systems sichtbar werden, wenn die Transaktion korrekt bis zum Ende geführt und bestätigt wurde (siehe unten).

Durability: Wenn eine Transaktion korrekt beendet wurde, muss sichergestellt werden, dass alle ihre Änderungen auch im Falle eines Systemausfalls dauerhaft im Datenbestand gespeichert werden.

Um die Einhaltung dieser Eigenschaften sicherstellen zu können, existieren in Transaktionssystemen häufig zwei Operationen zur Verwaltung der Transaktionen (vgl. [Bac98] S.438): Mit *Commit* wird eine Transaktion bestätigt, d.h. sie konnte ohne Fehler und Konsistenzverletzungen bis zum Ende ausgeführt werden, mit *Abort* lässt sich eine Transaktion abbrechen, falls Fehler oder Konsistenzverletzungen aufgetreten sind. Jede Transaktion muss entweder mit Commit oder Abort beendet werden. Dabei kann ein Abort entweder von der Transaktion selbst ausgelöst werden, falls ein Fehler auftritt, oder vom Transaktionssystem, falls sonst die oben genannten ACID-Eigenschaften nicht eingehalten werden könnten. Es müssen dann alle Veränderungen, die die Transaktion bis dahin an den Daten vorgenommen hat, rückgängig gemacht werden.

10.4.2 Transaktionen in XML-Prozessen

Im Zusammenhang mit XML-Prozessen liegen sogenannte *verteilte Transaktionen* vor. Dabei existiert nicht ein einzelner Datenbestand, sondern die Daten, auf denen die Operationen arbeiten, können sich auf verschiedenen Systemen befinden. Im Laufe einer Transaktion können dann Zugriffe auf Daten mehrerer Systeme stattfinden. Eine Möglichkeit zur Realisierung verteilter Transaktionen besteht in der Existenz eines zentralen Transaktionsverwalters, wie dies bei XML-Prozessen in Form des Prozessinterpreters der Fall ist. Innerhalb jedes beteiligten Systems gibt es jedoch einen lokalen Transaktionsverwalter, der für die Verwaltung der dort vorhandenen Daten zuständig ist. Die lokalen Verwalter werden vom zentralen Verwalter über Erfolg oder Scheitern von Transaktionen benachrichtigt und müssen dementsprechende Maßnahmen ergreifen (vgl. [Bac98] S.495ff).

Zusätzlich dazu ist bei XML-Prozessen die Situation gegeben, dass das Workflow-Management-System, d.h. der Prozessinterpreter, gar keinen direkten Zugriff auf alle Daten besitzt. Die einzigen Daten, von denen der Interpreter Kenntnis hat, ist das Migrationsdokument. Alle anderen Datenquellen werden ausschließlich innerhalb der Services eines Prozesses angesprochen, die für den Interpreter jedoch Black-Boxen darstellen. Der Interpreter kann also gar nicht selbst dafür sorgen, dass die ACID-Eigenschaften nicht verletzt werden. Weiterhin ist für das Migrationsdokument keine Transaktionskontrolle erforderlich, da niemals mehrere Services gleichzeitig darauf zugreifen und es zudem keinen persistenten Datenbestand im eigentlichen Sinne darstellt.

Die Rolle des Prozessinterpreters als Transaktionsverwalter muss sich aus diesen Gründen auf die Sicherstellung der Atomarität beschränken, indem er entweder im Falle eines Scheiterns des Prozesses alle betroffenen Systeme benachrichtigt, damit sie ihre Änderungen zurückführen oder ihnen mitteilt, dass die Transaktion bestätigt werden konnte. Er muss sich dann darauf verlassen, dass die Teilnehmer intern sicherstellen, dass die drei anderen Eigenschaften (Konsistenz, Isolation und Dauerhaftigkeit) eingehalten werden, beispielsweise indem sie selbst interne Transaktionen verwenden, um auf Daten von Back-End-Systemen zuzugreifen. Aufgrund dieser Gegebenheiten beschränken sich die folgenden Überlegungen ausschließlich auf die Einhaltung der Atomarität eines transaktionalen Prozesses.

Transaktionsfähige Services

Damit ein Prozess überhaupt als Transaktion ausgeführt werden kann, müssen alle verwendeten Services bestimmte Eigenschaften besitzen. Beispielweise müssen sie die Nachrichten über Erfolg oder Scheitern der Transaktion empfangen können oder - wie bereits erwähnt - so implementiert sein, dass sie Konsistenz, Isolation und Dauerhaftigkeit gewährleisten können. Sicherlich wird es nicht möglich oder erwünscht sein, dass alle Services diese Eigenschaften besitzen. Für Prozessdiagramme müsste daher die zusätzliche Bedingung formuliert werden, dass innerhalb von transaktionalen Prozessen ausschließlich geeignete Services verwendet werden dürfen. Falls diese Einschränkung nicht akzeptabel ist, könnte die Semantik eines transaktionalen Prozessdiagramms auch so definiert werden, dass nur die Änderungen von Services atomar ausgeführt werden, die Transaktionen unterstützen; alle anderen Services dürfen zwar ebenfalls verwendet werden, der Modellierer muss sich jedoch bewusst darüber sein, dass für sie die Atomarität nicht sichergestellt werden kann.

10.4.3 Fehler bei der Ausführung von Services

Prinzipiell sind zwei Arten von Fehlern denkbar, die dazu führen können, dass ein Prozess abgebrochen werden muss. Zum einen kann bei der Ausführung eines Service ein Fehler auftreten; dieser Fall soll in diesem Abschnitt betrachtet werden. Zum anderen kann ein Fehler oder Ausfall im Interpreter in seiner Rolle als Transaktionsverwalter auftreten (siehe Abschnitt 10.4.4).

Zur Behandlung von Fehlern, die während der Ausführung von Services auftreten, kann das Zwei-Phasen-Commit-Protokoll für verteilte Transaktionen verwendet werden (siehe [Bac98] S.502ff). Dabei sammelt der Prozessinterpreter als zentraler Verwalter die Ergebnisse aller Services (Phase 1). Nur wenn alle Services fehlerfrei ausgeführt wurden, kann die Transaktion bestätigt werden und die Commit-Entscheidung wird allen Services mitgeteilt (Phase 2). Falls jedoch Fehler aufgetreten sind, lautet die Entscheidung Abort, was ebenfalls allen Services mitgeteilt wird.

Maßnahmen zur Fehlerbehandlung

Es stellt sich nun die Frage, welche Bedeutung das Senden einer Commit- oder Abort-Nachricht an einen Service für sein Verhalten hat. Es gibt dafür prinzipiell zwei verschiedene Möglichkeiten:

1. Wenn der Service innerhalb eines transaktionalen Prozesses aufgerufen wird, merkt er sich zunächst nur intern alle Änderungen, die vorgenommen werden müssen. Falls er später eine Commit-Entscheidung vom Interpreter erhält, überträgt er alle Änderungen in den Datenbestand bzw. führt alle sonstigen notwendigen Aktionen aus. Im Falle eines Abort verwirft er einfach die gemerkten Änderungen.

2. Alternativ dazu könnte der Service zwar alle Aktionen bereits dann ausführen, wenn er im Prozess aufgerufen wird, dafür jedoch eine inverse Operation oder auch Kompensationsoperation anbieten. Falls später der Prozess aufgrund eines Abort zurückgesetzt werden muss, können mit Hilfe dieser Operation alle Aktionen rückgängig gemacht werden, so dass sich das System wieder im Ausgangszustand befindet. Als Beispiel ist ein Service denkbar, der einen Flug buchen soll. Der Buchungsvorgang muss nun nicht unbedingt verzögert werden, bis die Transaktion bestätigt wird, sondern es könnte stattdessen eine Stornierungsoperation zur Verfügung stehen, mit der sich die Wirkungen des Service aufheben lassen.

   Bei dieser Variante besteht das Problem, dass die Eigenschaft der Isolation nicht eingehalten werden kann, da möglicherweise Änderungen einer gescheiterten Transaktion im System sichtbar werden. Wenn dies vermieden werden soll, kann es vorkommen, dass auch andere Prozesse abgebrochen werden müssen, falls sie Daten gelesen haben, die von dem gescheiterten Prozess verändert wurden. Vergleiche hierzu das Problem der *kaskadierenden Abbrüche* (siehe [Bac98] S.450). Im Bereich von Workflows reichen laut Studien wie beispielsweise [Kon96] oder [Day91] häufig schwächere Anforderungen an Transaktionen als die herkömmlichen. Es wäre daher zu prüfen, ob die Verletzung der Isolation möglicherweise kein Problem darstellt und toleriert werden könnte.

Wir schlagen für XML-Prozesse vor, dass es den Implementierungen der Services überlassen werden sollte, welche der beiden Varianten zu bevorzugen ist. Je nach Aufgabe des Service kann eine Variante vorteilhafter oder sinnvoller sein als die andere.

Abschließend sei bemerkt, dass das Zwei-Phasen-Commit-Protokoll von der fehlerfreien Ausführung der Operationen Commit und Abort ausgeht, d.h. wenn ein Service bei seinem Aufruf keinen Fehler gemeldet hat, so darf später beim Ausführen von Commit oder Abort auch kein Fehler mehr auftreten (siehe [Bac98] S.501). Wenn dies nicht sichergestellt werden kann, müssen weitergehende Konzepte entwickelt werden, auf die im Rahmen dieser Arbeit jedoch nicht eingegangen werden kann.

Fehlerbehandlung durch den Prozess

Bei den bisherigen Überlegungen sind wir davon ausgegangen, dass ein transaktionaler Prozess abgebrochen werden muss, sobald in einem Service ein Fehler auftritt. Wie schon im Abschnitt 10.3.2 dargelegt, kann es jedoch Situationen geben, in denen die Behandlung von Fehlern explizit im Prozessmodell berücksichtigt werden soll, so dass es nicht zu einem Abbruch des Prozesses kommt. Diese Überlegungen lassen sich auch auf transaktionale Prozesse übertragen. Statt einen Fehler an den Prozessinterpreter zu melden, welcher die Transaktion mit einem Abort beenden würde, könnte ein Service auch eine entsprechende Fehlermeldung in das Migrationsdokument schreiben. Diese wird vom Prozess erkannt, so dass beispielsweise ein alternativer Service ausgeführt und so das Scheitern des Prozesses vermieden wird.

Trotzdem kann es bei der expliziten Modellierung von Fehlerbehandlungen Situationen geben, in denen eine weitere Ausführung nicht möglich ist. Dies könnte beispielsweise eintreten, falls alle alternativen Prozessstränge ausprobiert wurden und gescheitert sind. Es ist zu überlegen, ob für einen solchen Fall nicht ein neuer Zustandstyp in Prozessdiagrammen eingeführt werden sollte. Es könnte sich dabei um einen speziellen UML-Aktionszustand handeln, dessen Semantik so definiert ist, dass sofort die Ausführung des Prozesses beendet wird. Er stellt quasi einen Endzustand dar, der sich jedoch vom normalen Endzustand dadurch unterscheidet, dass er ein Scheitern des Prozesses signalisiert. Ein solcher Zustand würde nicht nur in transaktionalen Prozessen Sinn machen, sondern könnte auch im Zusammenhang mit Fehlerbehandlungen gemäß Abschnitt 10.3.2 zum Einsatz kommen.

10.4.4 Ausfall des Prozessinterpreters

Neben dem Auftreten von Fehlern innerhalb von Services besteht die zweite Art von Prozessfehlern in Ausnahmesituationen innerhalb des Prozessinterpreters in seiner Rolle als Transaktionsverwalter. Dabei sind gewöhnliche Laufzeitfehler relativ unkritisch, da der Interpreter lediglich so implementiert werden muss, dass dann die Transaktion mit Abort abgebrochen und dies allen betroffenen Services mitgeteilt wird. Dies ist mit Hilfe des Mechanismus der Java-Exceptions einfach möglich.

Als weitaus problematischer erweisen sich Ausfälle des Interpreters die durch einen Absturz der Systemumgebung hervorgerufen werden können. Entsprechend der Semantik von Transaktionen muss auch in einem solchen Fall die Einhaltung der ACID-Eigenschaften sichergestellt sein. Dabei besteht eine Reihe von Problemen, für die Lösungskonzepte entwickelt werden müssen.

1. Eine unerlässliche Maßnahme zur Behandlung von Systemausfällen ist die Protokollierung der Prozessausführung durch den Interpreter (vgl. [Bac98] S.484ff), damit er beim Wiederanlaufen des System ermitteln kann, an welcher Stelle eines Prozesses der Ausfall stattgefunden hat und dementsprechende Maßnahmen ergreifen kann.

2. Ein grundlegendes Problem besteht darin, dass nach einem Ausfall des Prozessinterpreters alle Objektinstanzen der Konnektoren, auf denen Services aufgerufen wurden, verloren sind. Nach dem Wiederanlaufen müsste der Interpreter sich neue Instanzen besorgen (vgl. Abschnitt 10.2.4). Dabei ist zu berücksichtigen, dass diese in der Lage sein müssen, die Arbeit der ursprünglichen Instanzen fortzuführen oder rückgängig zu machen.

3. Falls der Ausfall in der sogenannten Commit- oder Abort-Phase stattgefunden hat, d.h. während der Interpreter damit beschäftigt war, alle Services über Erfolg oder Scheitern des transaktionalen Prozesses zu benachrichtigen, kann das Drei-Phasen-Commit-Protokoll zum Einsatz kommen (siehe [Hei00B] 6-47ff). Es wurde dazu entwickelt, um speziell den Ausfall des Transaktionsverwalters in verteilten Systemen zu behandeln.

4. Für den Einsatz des Drei-Phasen-Commit-Protokolls ist es notwendig, dass sich alle Teilnehmer einer Transaktion - in unserem Fall die Services - untereinander kennen und kommunizieren können. Diese Fähigkeit wird dazu verwendet, um die Commit- oder Abort-Entscheidung, die einige Services schon erhalten haben könnten, bevor der Systemausfall stattgefunden hat, auch den übrigen mitzuteilen. Bei XML-Prozessen existiert bislang jedoch kein Mechanismus, mit dem die einzelnen Services eines Prozesses untereinander Nachrichten austauschen können.

5. Falls der Prozessinterpreter ausfällt, befinden sich möglicherweise die Back-End-Systeme, die durch Services angesprochen wurden, in einer Art Ungewissheitssituation, da sie auf ein Commit- oder Abort warten. Dieser Zustand kann je nach Gegebenheit nicht über längere Zeit toleriert werden, was vom Drei-Phasen-Commit-Protokoll durch *Timeouts* berücksichtigt wird. Dabei ginge ein Service nach Ablauf eines bestimmten Zeitintervalls davon aus, dass der Interpreter ausgefallen ist. Er muss dann im Rahmen des sogenannten *Terminierungsprotokolls* selbständig aktiv werden und in Kontakt mit den anderen betroffenen Services treten. Auch diese Forderung ist bei XML-Prozessen bislang nicht erfüllt, da Services stets passive Komponenten sind, die lediglich durch einen Methodenaufruf von außen angesprochen werden können.

Da wird auf die Implementierung von transaktionalem Verhalten verzichtet haben, möchten wir es bei der Aufzählung dieser Schwierigkeiten belassen und im Rahmen dieser Arbeit keine Lösungen für dieses recht komplexe Problemgebiet entwickeln.

10.5 Entwicklung von Konnektoren

Im Abschnitt 6.1.2 wurde erläutert, warum sich vorhandene Softwarekomponenten in der Regel nicht direkt in einen Workflow integrieren lassen. Als Lösung dieses Problems wurde das Konzept der Konnektoren vorgestellt, die als Bindeglied zwischen den Komponenten und dem Workflow-Management-System dienen. Sie stellen dem WFMS eine für alle Komponenten einheitliche Schnittstelle zur Verfügung. Bei XML-Prozessen besteht die Hauptaufgabe dieser Konnektorschnittstellen darin, das XML-Migrationsdokument entgegenzunehmen und Methoden für die einzelnen Services anzubieten. Abbildung 10.12 zeigt die dazu relevanten Klassen. Dabei sind alle vorhandenen Bestandteile weiß dargestellt, solche, die zur Anbindung einer Komponente neu zu implementieren sind, grau.

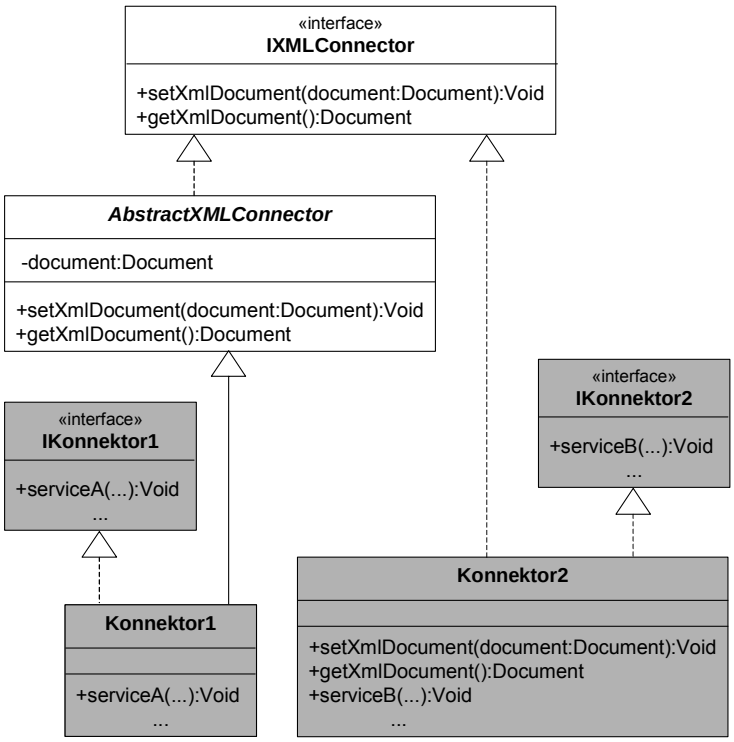


Abb. 10.12: Konnektor-Klassen

Zunächst gibt es das Interface *IXMLConnector*, das von allen Konnektoren erfüllt werden muss. Es fordert die Methode `setXmlDocument(..)`, mit der der Prozessinterpreter dem Konnektor das Migrationsdokument übergibt. Der Konnektor muss sich dieses Dokument intern merken, damit es von einem anschließend aufgerufenen Service verarbeitet werden kann. Wenn der Service beendet ist, ruft der Prozessinterpreter die Methode `getXmlDocument()` auf. Sie muss das vom Konnektor manipulierte Dokument liefern, das an den nächsten Prozesszustand übergeben werden kann. Als Datentyp für das Dokument wird das Interface *org.w3c.dom.Document* aus dem DOM-Modell des World Wide Web Konsortiums verwendet (siehe [DOM01]).

Vom Interface IXMLConnector abgeleitet ist die Klasse *AbstractXML-Connector*. Sie implementiert die beiden geforderten Methoden, indem sie sich intern das Dokument merkt. Es gibt nun zwei Möglichkeiten, eine Konnektor-Klasse zu realisieren. Die bequemste Variante ist es, diese von der Klasse AbstractXMLConnector

erben zu lassen. Sie muss dann lediglich die Service-Methoden enthalten, die vom Konnektor-Interface (hier: IKonnektor1 und IKonnektor2) gefordert werden. Sie haben über die geerbte Methode `getXmlDocument()` jederzeit Zugriff auf die XML-Daten (vgl. Konnektor1). Falls dies nicht möglich ist, weil beispielsweise die Konnektor-Klasse bereits eine Oberklasse besitzt, muss sie das Interface IXMLConnector direkt implementieren. Neben den Service-Methoden muss sie dann auch die beiden Methoden des Interface in geeigneter Weise realisieren (vgl. Konnektor2).

Zugriff auf die Prozessumgebung

Im Abschnitt 6.2.2 wurde beschrieben, dass es eine Prozessumgebung geben kann, in der sich zusätzliche Daten befinden. Es kann nun prinzipiell vorkommen, dass ein Konnektor nicht nur über ggf. vorhandene Service-Parameter, sondern direkten Zugriff auf diese Prozessumgebung benötigt. In einem solchen Fall muss der Konnektor zusätzlich zu IXMLConnector auch das Interface *IEnvironmentDependent* und damit die Methode `setEnvironment(..)` implementieren. Der Prozessinterpreter übergibt ihm dann vor Aufruf des Service mit Hilfe dieser Methode eine Instanz der Klasse *AbstractEnvironment*, die sich der Konnektor intern merken muss. Diese Klasse geht davon aus, dass sich in der Prozessumgebung eine Menge von XML-Dokumenten befindet, die sich jeweils durch einen eindeutigen Namen identifizieren lassen. Je nachdem in welcher Systemumgebung der Prozessinterpreter zum Einsatz kommt, muss eine konkrete Implementierung von AbstractEnvironment erstellt werden. Der Konnektor kann dann durch Aufruf der Methode `getXML(..)` auf die XML-Daten zugreifen. Vergleiche zur Prozessumgebung auch Abschnitt 10.6.2.

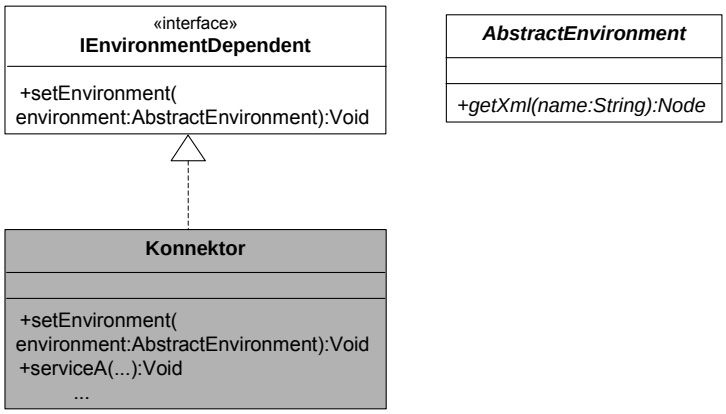


Abb. 10.13: Zugriff auf die Prozessumgebung

10.6 Integration in sunShine

Der Prozessinterpreter wurde zunächst als völlig selbständiges System entwickelt, so dass er prinzipiell in jeder Umgebung eingesetzt werden kann. Beispielhaft sollte er unter dem Namen *sunFlow* in die E-Business-Plattform sunShine integriert werden (vgl. Abschnitt 4.3). In diesem Abschnitt wird beschrieben, wie diese Integration vorgenommen wurde und welche Schritte dazu notwendig waren.

10.6.1 Der Sunflow-Transformer

Im Abschnitt 4.3 wurde das Konzept der Transformer von sunShine vorgestellt. Damit ist es möglich, eine Ressource zu definieren, die über eine URL bei einem Webserver angesprochen werden kann. Es wird dann ein XML-Dokument aus einer bestimmten Quelle gelesen, das anschließend mit beliebig vielen Transformern verarbeitet werden kann, bevor es an das anfragende Endgerät übermittelt wird. Wir haben uns entschieden, diese flexible Möglichkeit der XML-Verarbeitung zu nutzen, um den Prozessinterpreter in sunShine zu integrieren. Dazu wurde eine Klasse SunflowTransformer entwickelt, die für die Definition von Ressourcen verwendet werden kann.

Für die Auswahl des auszuführenden Prozesses sind zwei Möglichkeiten vorgesehen. Zum einen kann der Prozess direkt an die Ressource bzw. die URL gebunden sein. Er lässt sich dann beispielsweise durch Aufruf der URL `http://beispielserver.de/executeProcess1` ansprechen. Dies hat den Vorteil, dass der Webserver nur die Prozesse nach außen hin anbietet, für die eine Ressource definiert wurde. Zum anderen lässt sich der Prozess auch innerhalb der URL durch den Wert des Parameters „process“ auswählen. Der Aufruf könnte dann die Form `http://beispielserver.de/processinterpreter?process=Process1` haben. Bei dieser Variante ist jedoch zu beachten, dass alle Prozesse, die dem System bekannt sind, und nicht nur eine genau definierte Auswahl von außen aufgerufen werden können.

Als XML-Eingabedokument dienen die XML-Daten, die der SunflowTransformer beim Aufruf der Ressource vom Webserver erhält. Er übergibt die Daten an den Prozessinterpreter und führt den gewünschten Prozess aus. Das XML-Ausgabedokument wird dem Webserver als Ergebnis des SunflowTransformers übergeben. Da für die Ressource jeder beliebige Generator verwendet werden kann, ist als Herkunftsort des Eingabedokuments jede mit sunShine mögliche Datenquelle denkbar. Meistens wird es jedoch sinnvoll sein, dieses beim Aufruf der Ressource vom Client aus zu übergeben oder aus den übermittelten Daten eines Webformulars ein XML-Dokument zu generieren. Beide Varianten sind mit Hilfe von sunShine bereits realisierbar.

10.6.2 sunShine-Prozessumgebung

Im Abschnitt 10.5 wurde angedeutet, dass für jede Systemumgebung, in der der Prozessinterpreter eingesetzt werden muss, eine Unterklasse von AbstractEnvironment entwickelt werden muss, die die speziellen Gegebenheiten berücksichtigt. Zu diesem Zweck wurde zur Integration in sunShine eine Klasse *SunshineEnvironment* implementiert. Dies gestaltete sich relativ einfach, da mit den sogenannten *Sessioncontexts* bereits eine Menge von XML-Dokumenten zur Verfügung steht, die sich jeweils über einen

eindeutigen Namen ansprechen lassen (vgl. Abschnitt 10.5).

Eine Anforderung an die Integration des Interpreters in sunShine war es, dass sich die bisherigen Transformer als Softwarekomponenten in XML-Prozessen wiederverwenden lassen, da sie eine Menge von Standardlösungen für die Verarbeitung von XML-Daten darstellen. Zu diesem Zweck mussten eine Reihe weiterer sunShine-Komponenten über die Klasse SunshineEnvironment bekannt gemacht werden, da sie von den vorhandenen Transformern benötigt werden. Es ist uns auf diese Weise gelungen, tatsächlich jeden beliebigen Transformer mit Hilfe von Konnektoren als Aktivität in ein Prozessdiagramm integrieren zu können.

11 Evaluierung anhand einer Fallstudie

Zu Beginn der Diplomarbeit wurde in Abschnitt 3.1 eine Fallstudie durchgeführt, die die Anforderungen der Deutschen Bausparkasse Badenia AG an eine E-Business-Plattform behandelt, die in diesem Finanzdienstleistungsunternehmen installiert werden sollte. Die S&N AG hat unter Einsatz ihrer Integrationsplattform sunShine eine Lösung für dieses Problem entwickelt und implementiert. Da dabei allerdings nur sunShine in seiner bisherigen Form – also ohne die hier entwickelten Modellierungskonzepte und -werkzeuge – Verwendung fand, wollen wir diese Fallstudie noch einmal aufgreifen und untersuchen, ob die Lösung durch Einbeziehung der Konzepte dieser Arbeit noch verbessert werden könnte. Am Ende dieses Kapitels erfolgt schließlich eine Bewertung der Ergebnisse und Konzepte anhand der allgemeinen Anforderungen, die im Abschnitt 3.2 an ein System zur prozessorientierten Integration im E-Business gestellt wurden.

11.1 Vergleich der Lösungsansätze

Zur Erinnerung sei hier die Fallstudie aus Abschnitt 3.1 kurz zusammengefasst (für Details siehe dort): Gewünscht wurde als E-Business-Plattform ein Workflow-Management-System zur Verarbeitung von XML-Dokumenten. Zur Realisierung eines Multi-Channel-Ansatzes sollten verschiedene Ein- und Ausgabeformate unterstützt, aber die Dokumente jeweils in ein internes XML-Format transformiert werden. Eine Regelmaschine sollte je nach ausgewähltem Prozess und Zustand des Dokumentes dafür sorgen, dass das Dokument zur Weiterverarbeitung an den richtigen Nachfolgeservice weitergeleitet wird. Die vorhandenen Services sollten dann die nachgelagerten Back-End-Funktionalitäten aufrufen und so die vorhandenen Anwendungs- und Datenbanksysteme integrieren.

Die Realisierung der E-Business-Infrastruktur erfolgte im Rahmen eines Projektes der S&N AG. Bei der Beschreibung der umfangreichen Projektrealisierung wollen wir uns auf die Aspekte des Workflow-Managements konzentrieren und Randprobleme wie die kontinuierliche Archivierung von Zwischenergebnissen und die Transformation von und nach anderen Datenformaten zur Umsetzung des Multi-Channel-Ansatzes an dieser Stelle ausklammern. In den Projektanforderungen wurde explizit die Verarbeitung von XML-Dokumenten gewünscht. Daher eignet sich die in Abschnitt 4.3 vorgestellte, XML-basierte Software sunShine für diese Aufgabe sehr gut. Die Anbindung der Back-End-Systeme und auch der Archivierungsdatenbank konnte durch die Entwicklung passender sunShine-Transformerklassen (siehe Abschnitt 4.3.3) realisiert werden. Dadurch wurde eine einfache Integration der vorhandenen Systeme möglich. In diesem Abschnitt wollen wir die bisherige Lösung mit sunShine und die Modellierungsmöglichkeiten mit Prozessdiagrammen gegenüberstellen. Der Vergleich soll die Vorteile zeigen, die eine visuelle Modellierungssprache bietet. Dazu orientiert dich der Vergleich im Folgenden an den Basiskonstrukten für Workflow-Modelle: Sequenz, Prozessschachtelung, bedingte Verzweigung und Parallelität.

Sequenzen und Subprozesse

Zur sukzessiven Verarbeitung der XML-Dokumente wurde für sunShine eine Vielzahl kleiner Pipelines definiert, die in der Regel jeweils nur einen Verarbeitungsschritt des Gesamtprozesses abbilden. Durch eine regelbasierte Steuerung wurden die einzelnen Pipelines dann zum Gesamt-Workflow verknüpft.

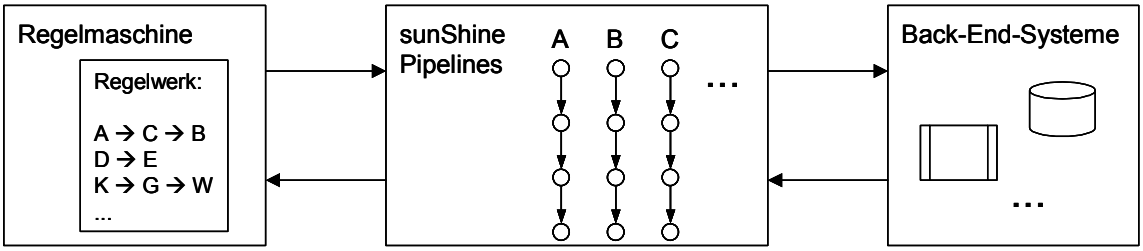


Abb. 11.1: Regelbasierte Workflow-Management-Architektur mit sunShine

Eine einzelne Pipeline erledigt normalerweise einen Prozessschritt, indem sie das aktuelle Dokument aus dem Archiv lädt, an einen Transformer zur Anbindung an die Back-End-Systeme weiterleitet, das Ergebnis gegebenenfalls durch die Anwendung eines Stylesheets transformiert und dann wieder im Archiv ablegt. Folgende Pipeline dient zum Beispiel zum Schreiben einer Prüfungsmeldung:

```
<match pattern="auditing-schreiben.xml">
      <generate src="auditing/auditing.xml"/>

      <!-- Aufruf des allgemeinen sunShine Transformers: -->
      <transform type="sunShine"/>

      <!-- Archiv-Transformers zum Laden des Dokuments: -->
      <transform type="archivTransformer"/>

      <!-- Transformation mit einem Stylesheet: -->
      <transform src="auditing/auditing-schreiben.xsl"/>

      <!-- Transformers zum Schreiben der Prüfungsmeldung: -->
      <transform type="auditingTransformer"/>

      <!-- Transformation mit einem Stylesheet: -->
      <transform src="auditing/auditing-ergebnis.xsl"/>

      <!-- Archiv-Transformers zum Sichern des Dokuments: -->
      <transform type="archivTransformer"/>

      <serialize type="xml"/>
</match>
```

Abb. 11.2: Pipeline-Beispiel (nach [Bad01])

Es sollen hier nicht die Details der Syntax von Pipelines erläutert, sondern vielmehr durch den Vergleich mit folgendem kleinem Prozessdiagramm die offensichtlichen Vorzüge der visuellen Modellierung verdeutlicht werden. Die gleiche Pipeline könnte als Prozessdiagramm grafisch modelliert etwa so aussehen:

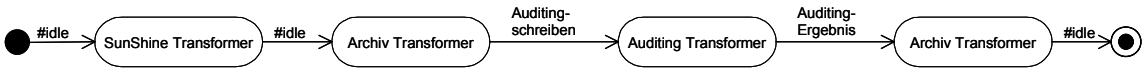


Abb. 11.3: Prozessdiagramm einer Pipeline

Je größer die Pipeline-Strukturen werden, desto mehr zahlt es sich aus, visuelle und damit übersichtlichere und besser handhabbare Modelle zu benutzen. Für komplexe Prozessmodelle scheinen sich die Pipelines gar nicht mehr zu eignen, weswegen bei der Projektlösung auch nur einzelne Prozessschritte durch Pipelines umgesetzt wurden. Um dennoch das geforderte Workflow-Management zu realisieren, musste man eine Regelmaschine als Java-Servlet-Applikation entwickeln, mit der die Zustandsübergänge zwischen den einzelnen Prozessschritten durchgeführt werden können. Die Regeln für diese Regelmaschine werden in einer eigens dafür entworfenen XML-Sprache beschrieben und entsprechen im Wesentlichen der Definition von Zuständen, in denen eine bestimmte Pipeline aufgerufen wird, und der Bestimmung von Folgezuständen. Ähnlich wie bei den Pipelines gibt es auch für die Definition der Regeln keine grafische Repräsentation und kein visuelles Modellierungswerkzeug, so dass die sehr technisch formulierten Regeln manuell vom Entwickler eingegeben werden müssen. Um einen kleinen Eindruck von der Syntax dieser Regelsprache zu vermitteln, soll hier ein kleiner, bereits vereinfachter Auszug gezeigt werden, ohne im Detail auf die Semantik einzugehen:

```
<Prozesstyp name="Auditing">

   <Zustand id="0" name="Auditing schreiben" start="Ja">
      <StandardAktion final="Nein" folgeid="1" name="Auditing">
         <!-- Verweis auf die Pipeline: -->
         <Url url="http://sunshine/auditing-schreiben.xml"/>
      </StandardAktion>
   </Zustand>

   <Zustand id="1" name="Auditing Antwort" start="Nein">
      <StandardAktion final="Ja" name="Auditing Antwort">
         <!-- Verweis auf die Pipeline: -->
         <Url url="http://sunshine/auditing-antwort.xml"/>
      </StandardAktion>
   </Zustand>

</Prozesstyp>
```

Abb. 11.4: Auszug aus dem Regelwerk zur Ablaufsteuerung (nach [Bad01])

Die Idee und Konzeption des Regelwerks in Verbindung mit der Regelmaschine erinnert an die Interpretation der PML-Dokumente (vgl. Kapitel 8) durch den in Kapitel 10 vorgestellten Prozessinterpreter. In der Tat gibt es einige Parallelen in den beiden zu Grunde gelegten XML-Formaten, allerdings besteht der wohl gravierendste Unterschied darin, dass mit PML die Prozessbeschreibungen einheitlich vorliegen und nicht in Regelwerk und Pipelines zerstückelt werden. Bei Prozessdiagrammen kann zwar auch durch hierarchische Anordnung von Subprozessen eine Gliederung vorgenommen werden, man bleibt aber in einem einheitlichen, homogenen Modell. Die oben abgedruckte Regel könnte zum Beispiel so modelliert werden:

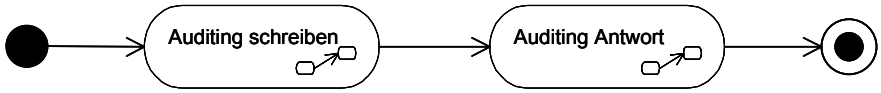


Abb. 11.5: Prozessdiagramm für eine Regel zur Ablaufsteuerung

In den Subaktivitätszuständen werden die Prozesse eingebunden, die bisher durch eine Pipeline (siehe Abbildungen 11.2 und 11.3) realisiert wurden. Durch dieses abgeschlossene Konzept können einige Nachteile der bisher vorgenommenen Fragmentierung in Pipelines und Regelwerk beseitigt oder zumindest abgemildert werden. Zu diesen Nachteilen der bisherigen Lösung zählen:

- eine schlechtere Kontrolle des Gesamtablaufs durch Verteilung der Ablaufsteuerung auf Regelmaschine und Pipeline-Ausführung,
- damit verbundene Einbußen bei der Ablaufgeschwindigkeit
- schwierigere Verwaltung und Wartung der Prozessbeschreibungen
- schwierigere Fehlerbehandlung und transaktionale Ausführung, weil die einzelnen Pipelines nichts voneinander wissen und es daher im Fall eines Abbruchs schwierig ist, den gesamten Prozess über den Abbruch zu informieren und zurückzusetzen.

Bedingte Verzweigungen

Zur Abbildung flexibler Geschäftsregeln in den Workflow-Modellen ist der Einsatz von bedingten Verzweigungen unverzichtbar. In dem Regelwerk der Badenia AG kann man dazu für jeden Zustand mehrere alternative Aktionen angeben und mit einem *Entscheidungswert* versehen. Es wird in einem Zustand dann nur diejenige Aktion ausgeführt, für die der Entscheidungswert wahr wird. Die Entscheidungswerte dürfen sich nicht überschneiden, so dass höchstens einer der Ausdrücke wahr werden kann. Außerdem kann eine Aktion als *Standardaktion* gekennzeichnet werden, die standardmäßig dann ausgeführt wird, wenn aufgrund der Entscheidungswerte keine der alternativen Aktionen durchgeführt werden konnte. Die Entscheidungswerte beinhalten triviale logische Ausdrücke, die sich jeweils auf den Rückgabewert beziehen, der von der Regelmaschine bei der Ausführung des vorausgegangenen Zustandes erzeugt wurde. Möglich ist die Angabe eines Zielintervalls oder eines Operators wie $<$, $\leq$, $=$, $\geq$, $>$ zusammen mit einem Operand als Vergleichswert. Wenn der Rückgabewert die (Un-)Gleichung erfüllt, wird die entsprechende Aktion ausgewählt.

Der folgende Auszug aus dem Regelwerk verdeutlicht die Anwendung der Entscheidungswerte an einem Beispiel zum Lesen von Kundeninformationen aus einer Datenbank. Die Regeln beschreiben einen Prozess, dessen zweite Aktion von dem Rückgabewert der ersten Aktion abhängt: In der ersten Aktion werden Kundendaten aus einer Datenbank eingelesen. Ist der Rückgabewert dieser Aktion gleich null, was einer fehlerfreien Ausführung entspricht, werden als nächstes die Kundendaten angezeigt, ansonsten (Standardaktion) wird eine Fehlerverarbeitung angestoßen.

```
<Prozesstyp name="Kundeninfo">

  <Zustand id="0" name="Start" start="Ja">
    <StandardAktion final="Nein" folgeid="1" name="Kundeninfo">
      <Url url="http://sunshine/kundeninfo-lesen.xml" />
    </StandardAktion>
  </Zustand>

  <Zustand id="1" name="Ergebnisbehandlung" start="Nein">
    <SyncAktion final="Ja" name="Kundendaten anzeigen">
      <Url url="http://sunshine/ergebnis-ok-kundeninfo.xml" />
      <Entscheidungswerte>
        <Ausdruck operator="eq" wert="0" />
      </Entscheidungswerte>
    </SyncAktion>
    <StandardAktion final="Ja" name="Fehlerverarbeitung">
      <Subprozess subprozess="Fehlerbehandlung"/>
    </StandardAktion>
  </Zustand>

</Prozesstyp>
```

Abb. 11.6: Regeln mit bedingter Verzweigung (nach [Bad01])

In Prozessdiagrammen werden solche bedingten Verzweigungen durch Entscheidungsausdrücke (guards) an den Transitionen beschrieben. Dieses Verfahren hat den Vorteil, dass dort beliebig komplexe boolsche Ausdrücke benutzt werden können. Außerdem muss die Ausführung einer Aktion nicht explizit einen Rückgabewert produzieren, der in den Guards abgefragt wird und so den weiteren Kontrollfluss steuert, sondern man kann in den Guards beliebige XPath-Ausdrücke angeben, um direkt auf die Inhalte des Migrationsdokuments oder Umgebungsvariablen zuzugreifen (vgl. Abschnitt 6.2.4). Dieses Konzept ist viel mächtiger, weil sich komplexere Entscheidungsbedingungen in Abhängigkeit vom Zustand des Migrationsdokuments formulieren lassen, als dies mit Rückgabewerten möglich ist. Im Übrigen lässt sich das Konzept der Rückgabewerte auf das Guard-Konzept abbilden, indem der Rückgabewert einer Aktion einfach im Migrationsdokument oder einer Umgebungsvariable abgelegt und in nachfolgenden Guards abgefragt werden kann. Das folgende Prozessdiagramm greift diese Methode auf und geht davon aus, dass der Rückgabewert im XML-Element <return> unterhalb der Wurzel des Migrationsdokuments abgespeichert ist. Dann stellt sich der obige Prozess mit seiner Verzweigung wie folgt als Prozessdiagramm dar:

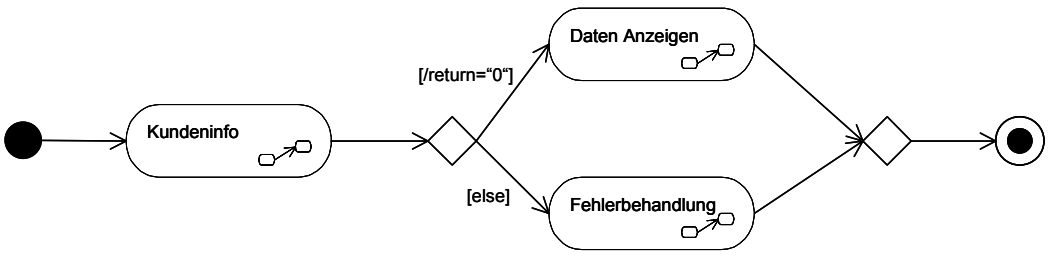


Abb. 11.7: Prozessdiagramm mit Verzweigung

Die Standardaktion aus dem Regelwerk wird im Prozessdiagramm durch den vordefinierten Guard mit der Bezeichnung „else“ (siehe [Omg01] 2-150f.) modelliert. Insge-

samt zeigt sich, dass Verzweigungen in der bisherigen Lösung nur über den relativ umständlichen Weg der Rückgabewerte spezifiziert werden können und somit auch bei diesem Kontrollflusskonstrukt Prozessdiagramme eine elegantere und mächtigere Option anbieten.

Parallelität

Zur Definition von Nebenläufigkeit wird im Regelwerk der E-Business-Lösung der Bausparkasse das Konstrukt der *asynchronen Aktionen* eingeführt. Dahinter verbergen sich Aktionen, die gleichzeitig zu der aufgrund der Entscheidungswerte ausgewählten *synchronen Aktion* ausgeführt werden sollen. Die Unterscheidung der beiden Aktionsarten erfolgt durch die XML-Elemente <SyncAktion> und <AsyncAktion>. In jedem Zustand kann also immer nur eine synchrone, aber durchaus mehrere asynchrone Aktionen durchgeführt werden. Ähnlich wie bei den synchronen Aktionen im vorausgegangenen Abschnitt vorgestellt, kann man auch den Start asynchroner Aktionen von der Erfüllung bestimmter Entscheidungswerte abhängig machen. Der folgende Auszug aus dem Regelwerk zeigt den kleinen Beispielprozess aus Abbildung 11.4, ergänzt um die parallele Ausführung einer zweiten Aktion, die nebenläufig eine Bestätigung des Prüfvorgangs (Auditing) absendet.

```
<Prozesstyp name="Auditing">

  <Zustand id="0" name="Auditing schreiben" start="Ja">
     <StandardAktion final="Nein" folgeid="1" name="Auditing">
        <Url url="http://sunshine/auditing-schreiben.xml" />
     </StandardAktion>
  </Zustand>

  <Zustand id="1" name="Auditing Antwort" start="Nein">
     <AsyncAktion name="Auditing Bestätigung">
        <!--Diese Aktion wird parallel ausgeführt-->
        <Entscheidungswerte>
           <Ausdruck operator="eq" wert="0" />
        </Entscheidungswerte>
        <Url url="http://sunshine/auditing-bestätigung.xml"/>
     </AsyncAktion>
     <StandardAktion final="Ja" name="Auditing Antwort">
        <Url url="http://sunshine/auditing-antwort.xml"/>
     </StandardAktion>
  </Zustand>

</Prozesstyp>
```

Abb. 11.8: Regeln mit paralleler Ausführung (nach [Bad01])

Genau wie in Prozessdiagrammen können beliebig viele Aktionen parallel durchgeführt werden, gegebenenfalls auch in Abhängigkeit von Entscheidungswerten (siehe oben). Diese Möglichkeit gibt es in Prozessdiagrammen durch die Verwendung von Guards an den ausgehenden Transitionen einer Parallel-Verzweigung (fork). Die folgende Abbildung zeigt die obige Regel in einer Darstellung als Prozessdiagramm.

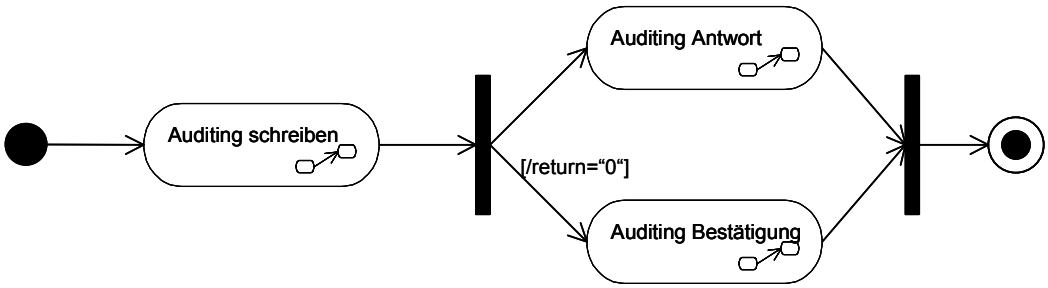


Abb. 11.9: Prozessdiagramm mit paralleler Ausführung

Die parallele Ausführung von Aktionen ist im Regelwerk leider immer nur für die Aktionen eines Zustands möglich. Der Wechsel in den nachfolgenden Zustand findet erst dann statt, wenn alle nebenläufigen Aktionen beendet worden sind. Es ist daher nicht möglich, ganze nebenläufige Prozessstränge mit mehreren Aktionen zu modellieren, die unabhängig voneinander ablaufen sollen. Weil dies in Prozessdiagrammen ohne Weiteres der Fall ist, bilden sie auch im Bereich Nebenläufigkeit das mächtigere und flexiblere Konzept. Zudem ist es bei Prozessdiagrammen möglich, Synchronisationspunkte einzubauen, um Abhängigkeiten zwischen parallelen Prozessteilen zu modellieren.

Somit bringt der Einsatz von Prozessdiagrammen und des zugehörigen Prozessinterpreters neben den Vorteilen der Veranschaulichung durch visuelle Modellierung weitere Verbesserungen gegenüber der bisherigen Lösung der Workflow-Management-Anforderungen der E-Business-Infrastruktur. Dazu gehören vor allem die größere Mächtigkeit und Flexibilität der Konstrukte der Modellierungssprache, die durch den Interpreter für Prozessdiagramme auch auf die Ausführungsebene übertragen werden. Vorgestellt wurden in diesem Abschnitt die Vorzüge einer einheitlichen, homogenen Prozessbeschreibung, der Möglichkeit für flexiblere bedingte Verzweigungen und der Definition umfangreicherer paralleler Teilabschnitte in Prozessdiagrammen im Gegensatz zu den bisher benutzen regelbasierten Beschreibungen. Auf die Umsetzung des gesamten Regelwerks aus der Fallstudie in Prozessdiagramme soll an dieser Stelle verzichtet werden; sie könnte analog zu dem gezeigten Beispielen auch für umfangreichere Prozesse aus der E-Business-Plattform des Finanzunternehmens vorgenommen werden.

11.2 Erfüllung der Anforderungen

Nach dieser Betrachtung der Anwendbarkeit in einem konkreten Fallbeispiel bleibt abschließend noch die Überprüfung, ob die zu Beginn der Arbeit gesetzten Anforderungen (vgl. Abschnitt 3.2) und Ziele (3.4) für ein Workflow-System im E-Business erreicht wurden.

Die Forderung nach einem Modellierungswerkzeug, das auch bei der Konsistenzprüfung unterstützt, kann mit dem entwickelten Prozesseditor erfüllt werden. Die Anbindung von Back-End-Systemen bei der Prozessausführung erfolgt durch dynamisch zu bindende Konnektoren, die gleichzeitig die Zuordnung von Workflow-Elementen auf reale Komponenten vornehmen. Die Verarbeitung von XML-Dokumenten und der Workflow-relevanten Daten kann durch das Migrationsprinzip und die Ausrichtung auf XML erfüllt werden. Mit PML liegt auch die Workflow-Beschreibung in einem XML-Format vor. Der entwickelte Prozessinterpreter interpretiert diese Beschreibungssprache und sorgt so für die Kontrolle und Ausführung der Prozess-

instanzen. Durch die vorhandenen XML-Technologien und die Integration in das sunShine E-Business-System werden Portalfunktionalität und Multi-Channel-Ansatz realisiert. Die Anforderungen bzgl. Benutzerinteraktion und Administrator-Schnittstelle wurden für die Ziele dieser Arbeit als weniger relevant angesehen (vgl. Abschnitt 3.4) und daher auch nicht oder nur teilweise gelöst. Konzepte für die Transaktionsunterstützung konnten aufgrund der begrenzten Zeit nur umrissen, aber nicht in die Tiefe gehend behandelt werden (vgl. Abschnitt 10.4). Durch die Verwendung der Prozessdiagramme können flexible Workflow-Modelle für die Geschäftsregeln der Abläufe aufgestellt werden. Die Implementierung in der Programmiersprache Java sichert die Plattformunabhängigkeit des Systems und damit die Einsetzbarkeit in verschiedenen, heterogenen Umgebungen. Die folgende Tabelle zeigt die Erfüllung der Anforderungen noch einmal im Überblick.

Anforderung	**sunShine**	**sunShine + sunFlow**
Modellierungswerkzeug	-	+
Analyse, Optimierung und Konsistenzprüfung	-	+
Anbindung von Back-End-Systemen	+	+
Workflow-relevante Daten	+	+
Verarbeitung von XML-Dokumenten	+	+
Workflow-Beschreibungen in XML	+	+
Zuord. von Workflow-Elementen auf reale Komp.	2)	+
Kontrolle und Ausführung der Prozessinstanzen	+	+
Portalfunktionalität	+	+
Multi-Channel-Fähigkeit	+	+
Benutzerinteraktion	1)	1)
Administrator-Schnittstelle	o	o
Flexibles Workflow-Modell für Geschäftsregeln	-	+
Transaktionsunterstützung	-	o
Plattformunabhängigkeit	+	+

1) Behandlung nicht implizit vorgesehen, aber mit vorhandenen Mitteln realisierbar.
2) Im Modell können nur bereits vorhandene Komponenten verwendet werden.

+ = vorhanden - = nicht vorhanden o = ansatzweise vorhanden

Tab. 11.1: Bewertung von bisherigem und neuem System im Vergleich

Zum Vergleich zeigt die Tabelle die Erfüllung der Anforderungen sowohl für das bisherige System sunShine der S&N AG, wie es in Abschnitt 4.3 vorgestellt wurde, als auch für das neue System sunFlow, das in sunShine integriert worden ist. Dadurch werden die erzielten Verbesserungen gegenüber der früheren Situation offensichtlich. Es zeigt sich, dass die wesentlichen Ziele und Anforderungen an die prozessorientierte Integrationsumgebung erfüllt werden konnten.

12 Schlussbetrachtungen

Im letzten Teil der Diplomarbeit wurde die praktische Umsetzung der entwickelten Konzepte dokumentiert. Damit schließt sich der Kreis von Anforderung, Konzeption und Umsetzung. Zum Schluss wollen wir in diesem Kapitel die erzielten Ergebnisse zusammenfassen und einen Ausblick auf die noch offenen Fragen geben.

12.1 Zusammenfassung

Ausgangspunkt in der Einleitung zur Diplomarbeit (vgl. Kapitel 1) waren die folgenden Problemstellungen und offenen Fragen bei der Softwareintegration im E-Business:

1. Die Anbindung heterogener Systeme an eine E-Business-Plattform und die Kopplung der Systeme untereinander.
2. Die Einbindung der betrieblichen Geschäftsprozesse und Workflows: Die vorhandenen Informationssysteme des Unternehmens sollten in den Aktivitäten der Workflows aufgerufen werden. Die bei diesem prozessorientierten Ansatz entstehenden elektronischen Geschäftsprozesse könnten über eine E-Business-Infrastruktur und das Internet von externen Partnern, Mitarbeitern und Kunden ausgelöst werden.
3. Die vollständige Automatisierung der Workflows durch Kontroll- und Datenfluss zwischen den benutzten Softwarekomponenten, die ohne Benutzerintervention ablaufen können. Dazu müssen Schnittstellen zum automatischen Datenaustausch zwischen den Applikationen geschaffen werden.
4. Die Schaffung flexibler Zugriffsmöglichkeiten für die Kunden mit ihren verschiedenen Endgeräten und damit der Unterstützung eines Multi-Channel-Ansatzes.
5. Die Unterstützung der Mitarbeiter eines Unternehmens, die sich mit der Softwareintegration befassen, bei ihren teilweise komplexen Aufgaben durch entsprechende Werkzeuge.

Im folgenden Rückblick wollen wir diese Problemstellungen und die einzelnen Ziele der Diplomarbeit rekapitulieren und die dazu im Hauptteil der Arbeit vorgestellten Lösungsoptionen und Konzepte reflektieren. Neben den in der Einleitung (Kapitel 1) aufgeführten Zielen werden auch die Anforderungen aus der Anforderungsanalyse (Kapitel 3) als Maßstab herangezogen. Diese fußt auf einer vorab durchgeführten Fallstudie (Abschnitt 3.1), in der Anforderungen einer großen deutschen Bausparkasse untersucht wurden, die in einem Pilotprojekt ihre Geschäftsprozesse über eine E-Business-Infrastruktur für Kunden und Außendienstmitarbeiter verfügbar machen und dabei ihre vorhandenen Informationssysteme einbinden möchte. Eingeflossen sind aber auch Anforderungen, die allgemein im Zusammenhang mit Workflow-Management im E-Business auftreten und im Kapitel 2 über die Grundlagen des Workflow-Managements betrachtet wurden.

Anhand der festgestellten Anforderungen erfolgte zunächst eine Evaluierung der vorhandenen Systeme Microsoft BizTalk, WARP und sunShine (Kapitel 4), um einen Einblick in den derzeitigen Stand der Technik zu geben. Dabei zeigte sich bei BizTalk

und WARP (siehe 4.1 und 4.2) ein großer Mangel bei der Unterstützung des Modellierers durch Konsistenzprüfungen zur Fehlervermeidung sowie der Realisierung eines Multi-Channel-Ansatzes und der Möglichkeit zur Integration in ein Unternehmensportal, was für den Einsatz im E-Business – zum Beispiel für virtuelle Marktplätze – sehr hilfreich wäre. Letzteres wird zwar vorbildlich von der E-Business-Plattform sunShine der S&N AG gelöst (siehe 4.3), sie weist aber andere Schwächen auf, die sich vor allem aus unzureichender Unterstützung bei der Modellierung flexibler Workflow-Modelle und dem vollständigen Fehlen eines visuellen Modellierungswerkzeugs ergeben. Eine Möglichkeit zur visuellen Modellierung ist jedoch Voraussetzung für die Einbeziehung informationstechnisch nicht geschulten Personals in das Workflow-Management (siehe 2.3). Somit hat sich gezeigt, dass bisher noch keine umfassende Lösung für die Anforderungen an eine prozessorientierte Integration von Softwarekomponenten im Bereich des E-Business existiert. Im zweiten Teil der Arbeit wurden darum eigene Konzepte entwickelt, die die Anforderungen erfüllen und einen prozessorientierten Ansatz zur Softwareintegration in E-Business-Systemen ermöglichen.

Erstes Ziel und eine wichtige Anforderung (siehe oben Punkt 1 und Abschnitt 3.2 Punkt 3) war die Einbindung bestehender Softwarekomponenten. Dazu wurde in Abschnitt 6.1 die Idee der Konnektoren vorgestellt. Dabei handelt es sich um kleine Softwaremodule, die eine Brückenfunktion zu den heterogenen Systemen übernehmen. Zur E-Business-Plattform bzw. zum Workflow-Interpreter bieten sie eine einheitliche Schnittstelle für den Datenaustausch und konvertieren diese Daten in Formate, die für die anzuschließenden Back-End-Systeme verständlich sind. Jeder Konnektor kann verschiedene Services anbieten. Jeder Service kapselt eine bestimmte Back-End-Funktionalität. Auf der Modellebene können die Services durch abstrakte Repräsentanten in die Workflow-Modelle als Akteure eingebunden werden. Bei der Ausführung der Modelle werden sie dann an reale Komponenten oder Objektinstanzen gebunden (siehe 3.2 Anforderung 7). Vorteil dieses Verfahrens ist die hohe Flexibilität durch die dynamische Bindung zur Laufzeit und die leichte Erweiterbarkeit, weil sich einfach neue Konnektoren in die Bibliothek der vorhandenen Konnektoren einfügen lassen, um so neue Back-End-Systeme zu integrieren.

Als weiteres Ziel der Diplomarbeit sollte untersucht werden, welche Möglichkeiten und Vorteile die Benutzung von XML-Dokumenten bringen könnte, weil XML gerade im Bereich E-Business weit verbreitet ist und sich als plattformneutrales Datenformat für den Datenaustausch in heterogenen Systemen gut eignet. Die Anforderungen der Fallstudie (siehe 3.1) haben die Praxisrelevanz von XML gezeigt: Dort wurde explizit der Einsatz von XML gefordert. Dieser Wunsch wurde auch für unseren Ansatz übernommen und ein Konzept XML-basierter Workflows entwickelt. Dabei werden die zuvor genannten Konnektoren als Akteure in Workflow-Modellen angeordnet, die jeweils ein XML-Dokument verarbeiten und das Ergebnis ihrerseits als XML-Dokument weiterreichen. Dies führt zum Konzept der Dokumentmigration (siehe 6.1), bei dem die Kontrollflüsse eines Workflows gleichzeitig den Datenfluss eines XML-Migrationsdokuments implizieren. Dieses Migrationsdokument enthält alle Workflow-relevanten Daten (siehe 3.2 Anforderung 4) und transportiert die Ergebnisse des einen Service zur Weiterverarbeitung an den nächsten Service. Dabei wird es sukzessive transformiert und mit Zwischenergebnissen angereichert. Das resultierende Dokument des letzten Verarbeitungsschrittes ist dann die Ausgabe des Workflows. Dieses Vorgehen vereinfacht die Handhabung der sehr umfangreichen Datenflüsse, indem alle für die

nachfolgenden Verarbeitungsschritte relevanten Zwischenergebnisse der Services durch das Migrationsdokument aufgenommen und mit dem fortschreitenden Kontrollfluss weitergeleitet werden (siehe oben Punkt 3). Außerdem bietet XML eine hervorragende Grundlage zur Realisierung des geforderten Multi-Channel-Ansatzes (siehe oben Punkt 4 und Abschnitt 3.2 Anforderung 9), weil sich mit Stylesheets die Ausgabedokumente der Prozesse leicht umstrukturieren und in die gewünschten Zielformate transformieren lassen.

Um das Ziel einer möglichst gut verständlichen, intuitiven Modellierung der Workflows zu erreichen, bedarf es einer visuellen Modellierungssprache (siehe 3.2 Anforderung 1). Wegen der hohen Bedeutung, die der Workflow-Modellierung als hilfreiches Abstraktionsmittel zum Entwurf und der Wartung von Workflows zukommt (siehe 2.3), enthält Abschnitt 3.3 eigene Anforderungen an eine solche grafische Workflow-Modellierungssprache. Die daraufhin in Kapitel 5 untersuchten, bekannten Modellierungssprachen wie Petri-Netze, Ereignisgesteuerte Prozessketten und UML-Aktivitätendiagramme genügten den zuvor aufgestellten Anforderungen jedoch nur eingeschränkt und besonders bei Petri-Netzen und den EPK zeigten sich Schwächen hinsichtlich der leichten Verständlichkeit oder der Möglichkeit zur Integration vorhandener Softwarekomponenten. Weil UML-Aktivitätendiagramme die Anforderungen am besten erfüllen und sie außerdem als Standard in Industrie und Wissenschaft anerkannt sind, wurden sie als Grundlage für die Modellierungssprache ausgewählt und unter Ausnutzung der UML-Erweiterungsmechanismen zur Umsetzung der XML-basierten Workflow-Modelle erweitert (siehe Kapitel 6 und 7).

Um die neuen Konzepte wie Dokumentmigration und Komponentenintegration in die Modellierungssprache zu integrieren, wurden Stereotypen zur Erweiterung des UML-Metamodells definiert, um Syntax und Semantik der Aktivitätendiagramme anzupassen (siehe 7.3). Daraus entstand ein eigenes UML-Profil für *Prozessdiagramme* zur Modellierung der XML-basierten Workflows. Dabei repräsentieren die Interfaces im statischen Modell die verfügbaren Konnektoren mit ihren Services zur Integration der vorhandenen Softwarekomponenten. Diese können in den Aktivitäten des dynamischen Modells aufgerufen werden. Dem Migrationsmodell entsprechend implizieren die Transitionen zwischen den Zuständen gleichzeitig die Überleitung des XML-Migrationsdokuments.

Außerdem wurde ein Konzept zur Modellierung der Dokumenttypen ausgearbeitet und in die Modellierungssprache bzw. das Metamodell für Prozessdiagramme integriert (siehe 6.3). Die Modellierung der Dokumenttypen durch die sogenannten *Ports* leistet einen bedeutenden Beitrag zur Konsistenzerhaltung und Fehlervermeidung bei der Modellierung: Mit diesem Typsystem können an den Ein- und Ausgabeschnittstellen der Services die Dokumenttypen spezifiziert werden, die der Service als Eingabe verarbeiten bzw. als Ausgabe produzieren kann. Werden die Services miteinander zu einem Workflow verkettet, muss sichergestellt sein, dass die Ausgabe eines Service als Eingabe des nachfolgenden Service benutzt werden kann. Wenn im Modell die entsprechenden Ports bereits zueinander kompatibel sind, kann auch die Kompatibilität der Dokumente zur Ausführungszeit gewährleistet werden. Die dem Port-Konzept zu Grunde liegende Theorie erlaubt eine algorithmische Suche nach inkonsistenten Zustandsübergängen bei inkompatiblen Port-Schnittstellen in einem Prozessdiagramm. Bei entsprechender Tool-Unterstützung kann der Modellierer dadurch von vornherein an der Erstellung fehlerhafter Modelle gehindert und bei der Modellierung gültiger

Modelle unterstützt werden (siehe oben Punkt 5). Der Zusatzaufwand bei der Modellierung, der sich aus der Berücksichtigung der Dokumenttypen und den Ports ergibt, kann durch den Algorithmus für die automatische Portberechnung (siehe 7.6) auf ein Minimum reduziert werden.

Wenn in einem Prozessdiagramm zwei Services durch eine Transition miteinander verknüpft werden sollen, deren Ein- und Ausgabeschnittstellen entsprechend ihrer Ports nicht zueinander kompatibel sind, kann die gewünschte Transition mit einem Stylesheet versehen werden. Die Anwendung des Stylesheets auf das Migrationsdokument transformiert dieses derart, dass es doch noch konform zur Zielschnittstelle wird. Somit wird einer Transition im Prozessdiagramm neben dem passiven Zustandsübergang zusätzlich ein aktives Verhalten zugeordnet. Vorteil dieser Methode ist eine viel größere Flexibilität bei der Verkettung von Services in den Workflows: Durch die transformierenden Transitionen können auch Services hintereinander ausgeführt werden, deren Schnittstellen eigentlich nicht kompatibel zueinander sind.

Um das Stylesheet bei ähnlichen Konstellationen einfach wiederverwenden zu können, wurde ein Konzept für typisierte Transitionen eingeführt, nach dem jeder Transition statt des Stylesheets ein wiederverwendbarer Transitionstyp zugeordnet wird, der seinerseits auf ein entsprechendes Stylesheet verweist. Die typisierten Transitionen und Transitionstypen wurden in das formale Konzept für die Ports aufgenommen, so dass zur Konsistenzprüfung nun eine einheitliche Theorie zu Grunde liegt, die auch die Kompatibilität bei Zustandsübergängen und eventuell vorgenommenen Transformationen bei Transitionen berücksichtigt. Durch die Typisierung der Transitionen wird dank hoher Wiederverwendbarkeit der Modellierungsaufwand reduziert.

Die textuell codierte Speicherung der Prozessmodelle (gemäß 3.2 Anforderung 6) erfolgt durch die Erzeugung einer Prozessbeschreibungsdatei in der XML-Sprache PML (Process Markup Language), die eigens zur textuellen Repräsentation von Prozessdiagrammen in Abschnitt 8.3 definiert wurde. Im Gegensatz zu anderen, in Abschnitt 8.2 vorgestellten XML-basierten Ansätzen wie XLANG und XMI erlaubt PML das Abspeichern der vollständigen Prozessdiagramme in einem für menschliche Benutzer gut lesbaren Format, ist für spezielle Anwendungen und Werkzeuge flexibel erweiterbar und kann direkt als Grundlage für die automatische Ausführung der Prozesse benutzt werden, wie dies bei dem in Abschnitt 10.2 vorgestellten Prozessinterpreter der Fall ist. Vorteil der XML-basierten Speicherung ist die Möglichkeit zur Konvertierung mit Stylesheets in andere Formate wie XMI zur Verarbeitung mit anderen Werkzeugen sowie eine gute Austauschbarkeit über das Internet.

Im dritten Teil der Diplomarbeit wurden die Resultate der praktischen Umsetzung dokumentiert. Entstanden ist dabei ein ganzheitliches System zur Modellierung und Ausführung der XML-basierten Workflows: Zur Erstellung und Verwaltung von Prozessdiagrammen wurde ein grafisches Modellierungswerkzeug (siehe Kapitel 9) entwickelt, mit dem sich die statischen und dynamischen Teile der Prozessmodelle spezifizieren lassen. Ein Algorithmus zur Validierung von Prozessmodellen (siehe Abschnitt 7.5), der in das Werkzeug integriert wurde, zeigt dem Benutzer Verletzungen der Wohlgeformtheit und Inkonsistenzen an und unterstützt ihn so bei der Integritätswahrung. Dabei greift der Algorithmus auf das entwickelte und ebenfalls in das Modellierungswerkzeug integrierte Port-Konzept zurück. Der im Hintergrund arbeitende Algorithmus zur automatischen Portberechnung reduziert zusätzlich den Mehraufwand des Benutzers zur Modellierung der Ports auf ein Minimum.

Im Prozesseditor werden die Prozessmodelle intern in einer Java-Implementierung des Metamodells für Prozessdiagramme verwaltet (siehe 7.4). Diese direkte Umsetzung der Metaklassen aus dem Metamodell (siehe 7.1) in Java erlaubt eine effiziente Arbeit der Werkzeuge auf den Prozessmodellen. Ein ebenfalls entwickelter PML-Exporter bzw. -Importer (siehe 8.4) dient zur wechselseitigen Konvertierung zwischen PML-Beschreibungen und der Metamodellimplementierung.

Die in PML vorliegenden Prozessbeschreibungen können von dem in Abschnitt 10.2 vorgestellten Interpreter eingelesen und auf Basis einer Zustandsmaschine ausgeführt werden. Er nimmt dabei eingehende XML-Dokumente an, leitet sie nach dem Migrationskonzept entsprechend der Prozessbeschreibung an die jeweiligen Konnektoren zur Verarbeitung weiter, nimmt sie nach jedem Verarbeitungsschritt wieder entgegen und fährt so lange fort, bis der gesamte Prozess abgearbeitet ist und das Ergebnis als XML-Dokument zurückgegeben werden kann. Der Prozessinterpreter setzt somit die Workflow-Modelle um und dient als Bindeglied zwischen der E-Business-Plattform und den nachgelagerten Back-End-Systemen, die in den Aktivitäten der Workflows vom Prozessinterpreter aufgerufen werden. Zusätzlich wurden in Abschnitt 10.3 Konzepte zur Fehlerbehandlung und in 10.4 zur transaktionalen Ausführung von Prozessen diskutiert.

Die implementierten Werkzeuge umfassen insgesamt 193 Java-Klassen mit etwa 30.000 LOC (lines of code). Sowohl bei der Software-Modellierung als auch bei der Implementierung hat sich der intensive Gebrauch von Entwurfsmustern bewährt. So konnte bei der Konstruktion des Datenmodells, dem flexiblen Zugriff über Schnittstellen auf die Datenstrukturen, der Realisierung der grafischen Benutzungsoberfläche und an vielen anderen Stellen auf Standardlösungen und bewährte Programmiererfahrung in Form von Entwurfsmustern zurückgegriffen werden. Im Rahmen der Zusammenarbeit mit der S&N AG in Paderborn wurden die so implementierten Werkzeuge in die E-Business-Plattform sunShine des Unternehmens integriert. Bisher war es dort nicht möglich, Workflows graphisch zu modellieren und dann automatisiert auszuführen. Statt dessen müssen die in Abschnitt 4.3 vorgestellten Pipelines manuell wie mit einer Programmiersprache kodiert werden. Die zu Anfang betrachtete und am Schluss aufgegriffenen Fallstudie (siehe Kapitel 11) hat gezeigt, welches Potenzial in den entwickelten Konzepten und Werkzeugen steckt. Die Konzepte der Arbeit wurden bei der Implementierung vollständig umgesetzt, so dass die erstellte Software nach geringen Optimierungen in realen Projekten und Produkten eingesetzt werden kann. Die Resonanz der Projektingenieure der S&N AG, die im Bereich E-Business tätig sind, war durchweg positiv und lässt in der Zukunft eine feste Integration der in dieser Arbeit entwickelten Konzepte in ihre Entwicklungsprozesse erwarten. Da sich die Werkzeuge zur Modellierung und Ausführung besonders für die einfachere Entwicklung flexibler Pipelines eignet, ist eine Integration in das Open-Source-Projekt Cocoon (siehe [Apa01]) der Apache-Organisation denkbar, weil auch dieses Projekt auf der erwähnten Pipeline-Architektur basiert. In jedem Fall sind die gewonnenen Ergebnisse von solcher Praxisrelevanz, dass sie in Zusammenarbeit mit der S&N AG weiterentwickelt und entsprechend der im folgenden Ausblick angedeuteten Ideen fortgedacht werden sollen.

12.2 Ausblick

Aufgrund des beschränkten Zeitrahmens einer Diplomarbeit konnten wir leider nicht alle Ideen und Aspekte erschöpfend betrachten und umsetzen. Im Folgenden soll eine Reihe solcher offenen Fragen angedacht und erste Überlegungen dazu angestellt werden. Es handelt sich aber nicht um fertige Konzepte, sondern lediglich um Ideen, wie eine mögliche Lösung aussehen könnte. Dabei geht es neben Optimierungen der bestehenden Konzepte auch um denkbare Erweiterungen sowie Möglichkeiten zur Verbesserung oder Effizienzsteigerung der implementierten Systeme.

Visualisierung von Ports: Bei der bisherigen Notation der Ports von verschiedenen Bestandteilen eines Workflow-Modells handelt es sich um eine rein textbasierte Darstellungsform. Im Abschnitt 6.3.1 wurde daher vorgeschlagen, Ports nicht innerhalb eines Prozessdiagramms darzustellen, sondern es einem CASE-Tool zu überlassen, sie dem Benutzer anzuzeigen. Es scheint uns schwierig zu sein, eine grafische Repräsentation für Ports zu entwickeln, da es sich um Mengen von XML-Elementen handelt, die ihrerseits textuell beschrieben werden. Denkbar ist eine große Menge verschiedener Ports oder Dokumenttypen, so dass sich nicht einfach eine überschaubare Menge von Farben oder Symbolen verwenden lässt, um Ports zu veranschaulichen. Sollte es dennoch gelingen, eine Möglichkeit zu finden, um Ports grafisch darzustellen, wäre dies ein großer Vorteil, da dem Modellierer die Ports in einer übersichtlichen und leicht verständlichen Weise präsentiert würden. Ziel für ein solches Konzept sollte es sein, während des Modellierungsvorgangs auf einfache Weise zu erkennen, welche Ports verkettet werden können, d.h. kompatibel zueinander sind.

Ausführung transaktionaler Prozesse: Im Abschnitt 10.4 wurden Vorschläge gemacht, wie sich Prozesse als Transaktionen ausführen lassen. Diese Fähigkeit ist bereits in der Modellierungssprache für Prozessdiagramme enthalten, lediglich der von uns implementierte Prozessinterpreter bietet dafür noch keine Unterstützung.

Pufferung von Prozessmodellen: Die Verarbeitung großer Mengen von XML-Daten beansprucht zur heutigen Zeit noch vergleichsweise viel Rechenzeit. Vor allem für Prozessbeschreibungen, die in Form von PML-Dokumenten vorliegen, wäre es daher sinnvoll, Pufferungsverfahren einzusetzen, damit vom PML-Importer geladene Prozesse nicht nur einmal, sondern für mehrere Ausführungen oder sogar vom Prozesseditor verwendet werden können (vgl. Abschnitt 10.2.1). Dabei sind unter anderem Überlegungen anzustellen, wie lange Prozessmodelle im Puffer gehalten werden sollen und wie mit Aktualisierungen im laufenden Betrieb umgegangen wird, d.h. falls ein Prozess im Editor verändert wird, während er gerade von einer Instanz des Interpreters ausgeführt wird.

Dynamisches Suchen von Konnektorinstanzen: Aus Sicht des Prozesseditors handelt es sich bei Konnektoren und Services ausschließlich um Interfaces und ihre Methoden, d.h. es sind lediglich Schnittstellen und nicht konkrete Implementierungen bekannt. Die Vorteile dieses Ansatzes wurden in 6.1.3 dargelegt. Allerdings hat dieses Vorgehen zur Folge, dass auch in der PML-Beschreibung eines Prozesses keine Klasse verankert ist,

die die benötigte Funktionalität anbietet, sondern es die Aufgabe des Prozesseditors ist, zur Ausführungszeit eine geeignete Klasse zu finden. Im Abschnitt 10.2.4 wurde erwähnt, dass bislang eine recht einfache Variante zum Einsatz kommt, bei der in einer XML-Datei jedem Interface, das in den Prozessdiagrammen als Konnektor verwendet wird, eine implementierende Klasse zugeordnet ist. Als Alternative ist es auch denkbar, dass statt einer solchen festen Zuordnung der Interpreter dynamisch zur Laufzeit anhand der Schnittstelle, die durch das Interface spezifiziert wird, eine passende Klasse sucht.

Seit einiger Zeit existiert die von Sun Microsystems entwickelte Technologie *Java Intelligent Network Infrastructure* (Jini). Laut [Jin01] ist Jini entwickelt worden, um eine Infrastruktur für dynamische Dienste in Netzwerken zur Verfügung zu stellen. Ein Dienst ist dabei als Software- oder auch Hardwarekomponente mit einer bestimmten Funktionalität zu verstehen. Dienste können sich selbstständig beim Jini-System an- und abmelden. Bei der Anmeldung werden sie in einem speziellen Verzeichnis registriert, um ihre Funktionen anderen Teilnehmern des Systems anzubieten. Wenn ein Teilnehmer einen Dienst in Anspruch nehmen möchte, so spezifiziert er diesen in Form eines Interface, das der Dienst erfüllen soll und übermittelt diesen Wunsch an das Jini-System. Anhand des Verzeichnisses aller zur Zeit angemeldeten Dienste wird nun ein passender Dienst gesucht. Bei erfolgreicher Suche wird eine Verbindung zwischen Dienst und Dienstnehmer hergestellt, damit die Interaktion durchgeführt werden kann.

Diese Möglichkeiten der Jini-Technologie können für das Suchen von Konnektor-Klassen durch den Prozessinterpreter genutzt werden. Alle Klassen, die vom Workflow-System genutzt werden sollen, müssten sich dann beim Jini-System anmelden, so dass sie im Verzeichnis vermerkt sind. Wenn nun eine Interpreterinstanz zur Ausführung eines Service einen Konnektor benötigt, kann anhand des Verzeichnisses eine passende Klasse ermittelt werden, die das im Prozessmodell verwendete Interface erfüllt. Da die Konzepte von XML-Prozessen und der Jini-Technologie offenbar gut zusammenpassen, sollte eine Implementierung des Ansatzes keine großen Probleme bereiten.

Unterstützung anderer Modellierungswerkzeuge: Prozessdiagramme als Sprache zur Modellierung von XML-Prozessen lassen sich, wie in Kapitel 7 gezeigt, vollständig als Erweiterung der UML beschreiben. Wir haben den Prozesseditor entwickelt, um dem Modellierer eines Workflows ein speziell auf Prozessdiagramme zugeschnittenes Werkzeug zur Verfügung zu stellen, mit dem sich solche Diagramme komfortabel entwerfen lassen. Prinzipiell ließen sich aber mit jedem beliebigen UML-Werkzeug Prozessdiagramme erstellen, sofern es die gesamten Möglichkeiten von UML-Aktivitätendiagrammen und der Erweiterungsmechanismen, insbesondere das Konzept der Stereotypen, unterstützt. Es ergibt sich dabei das Problem, dass der Prozessinterpreter ausschließlich Prozesse ausführen kann, die in Form einer PML-Beschreibung vorliegen. Es kann jedoch nicht davon ausgegangen werden, dass das UML-Werkzeug, mit dem ein Prozess modelliert wurde, als Ausgabe ein PML-Dokument erzeugen kann. Mit XMI (siehe Abschnitt 8.2.2) existiert ein Industriestandard, der als einheitliches Format zum Austausch von UML-Modellen zwischen verschiedenen Modellierungssystemen dient. Im Rahmen der Verbreitung von XMI nutzen immer mehr UML-Werkzeuge diese Sprache zur Speicherung von UML-Diagrammen. Wenn ein Prozessdiagramm also mit einem anderen Werkzeug als dem Prozesseditor entworfen wurde, könnte es zunächst in Form eines XMI-Dokuments abgelegt werden. Man benötigt dann eine

Konvertierungsmöglichkeit zwischen den beiden Formaten XMI und PML. Da es sich bei beiden um XML-Sprachen handelt, liegt es nah, zu diesem Zweck XSL-Stylesheets zu verwenden. Man müsste ein Stylesheet entwickeln, das aus einer XMI-Datei ein PML-Dokument konstruiert, welches dann als Eingabe für den Interpreter verwendet werden kann. Ebenso könnte es sinnvoll sein, ein zweites Stylesheet zu realisieren, das ein PML-Dokument in eine XMI-Datei übersetzt. Dies würde es ermöglichen, ein Prozessdiagramm, das mit Hilfe des Prozesseditors erstellt wurde, in einem anderen UML-Werkzeug zu bearbeiten.

Quellcodegenerierung: Als Alternative zur Ausführung von Prozessdiagrammen durch den Prozessinterpreter ist es vorstellbar, aus einem Diagramm eine Java-Klasse zu generieren, die eine automatisch erzeugte Implementierung des modellierten Prozesses darstellt. Es bietet sich an, als Konzept für die generierte Klasse auf das Prinzip einer Zustandsmaschine zurückzugreifen. Es könnte beispielsweise ein XSL-Stylesheet entwickelt werden, das ein PML-Dokument in Java-Quellcode transformiert, der dann nur noch übersetzt werden müsste. Prinzipiell ist dies mit der Sprache XSL möglich. Es ist zu untersuchen, inwieweit ein solches Vorgehen zu Effizienzsteigerungen gegenüber dem Konzept der Ausführung von Prozessen durch einen Interpreter führt.

Unterstützung von Benutzerinteraktion: Im Rahmen dieser Arbeit haben wir uns auf die Untersuchung rein systemorientierter Workflows beschränkt (vgl. Abschnitt 2.2.2), bei denen als Akteure ausschließlich Softwarekomponenten bzw. –systeme zum Einsatz kommen. Je nach Einsatzgebiet unterstützen verfügbare Workflow-Management-Systeme aber auch die Interaktion mit menschlichen Teilnehmern. Es ist zu untersuchen, inwieweit im Bereich des E-Business solche Möglichkeiten von Bedeutung sind. Denkbar ist beispielsweise, dass innerhalb eines XML-Prozesses an bestimmten Stellen Rückfragen an den Benutzer gestellt werden müssen, der die Ausführung des Prozesses veranlasst hat. Es müssten Konzepte entwickelt werden, wie sich solche Situationen in Diagrammen modellieren lassen und wie sie technisch umgesetzt werden können.

Ereignisbehandlung: Bislang stellt jeder XML-Prozess einen in sich geschlossenen Workflow dar, dessen Instanzen sukzessive ausgeführt werden. Es ist jedoch keine Kommunikation mit anderen Prozessen vorgesehen. Auch ein Austausch von Nachrichten zwischen den Aktivitäten einer Prozessinstanz ist nur mit Hilfe des Migrationsdokuments realisierbar. Beide Varianten lassen sich zwar ohne Weiteres innerhalb der aufgerufenen Services realisieren, die Modellierung eines solchen Informationsaustauschs im Prozessdiagramm ist jedoch nicht möglich. Falls dies in bestimmten Situationen wünschenswert wäre, könnte auf das Konzept der Ereignisse, die in UML-Aktivitätendiagrammen existieren, zurückgegriffen werden. Es ist zu untersuchen, wie sich dieses Konzept für die Gegebenheiten von XML-Prozessen anpassen lässt. Wenn Prozessdiagramme um eine solche Möglichkeit zur expliziten Beschreibung von Ereignissen erweitert werden, müsste voraussichtlich eine Art Ereignisbehandlung in den Prozessinterpreter integriert werden, über den der Informationsaustausch zwischen Prozessinstanzen stattfinden kann. Vergleiche dazu auch [Wie01].

Statt das bisherige Konzept zu erweitern, könnte auch eine spezielle Ereigniskomponente implementiert werden, die entsprechende Services anbietet, um Ereignisse auszulösen oder zu verarbeiten. Gegenüber der Methode, Informationen intern in den

Services auszutauschen, hat dies den Vorteil, dass im Prozessdiagramm sofort ersichtlich wird, an welchen Stellen Ereignisse verarbeitet werden.

Einsatz als Infrastruktur für Webservices: Mit dem Simple Object Access Protocol (SOAP, siehe [SOAP01]) scheint eine Technologie zu existieren, die sich aufgrund ihrer Möglichkeiten zu dem Standard bei der Kommunikation zwischen Softwaresystemen in heterogenen Netzwerken, insbesondere dem World Wide Web entwickeln wird. SOAP ermöglicht es laut [SOAP01], plattformunabhängig Informationen auszutauschen, ohne Rücksicht auf systemspezifische Eigenschaften der beteiligten Kommunikationspartner zu nehmen, indem XML als gemeinsame Sprache verwendet wird. Mit Hilfe von SOAP lassen sich sogenannte Webservices (siehe [Bet01]) realisieren, die bestimmte Funktionen im Netz zur Verfügung stellen. Diese können von beliebigen, ebenfalls SOAP-fähigen Systemen in Anspruch genommen werden. Diese Form der Koppelung von Softwaresystemen vermeidet eine Reihe von Problemen, die sich bei der Benutzung alternativer Technologien wie Java-RMI oder CORBA ergeben, so dass man hier große Erfolgsaussichten sieht. Es lohnt sich daher zu untersuchen, auf welche Weise sich XML-Prozesse mit den Möglichkeiten von SOAP verbinden lassen. Auf den ersten Blick sind hier zwei verschiedene Ansätze denkbar. Zum einen könnte man nach einer Möglichkeit suchen, Webservices als Aktivitäten innerhalb von Prozessdiagrammen zu verwenden. Da sowohl XML-Prozessen als auch Webservices auf XML als Datenformat zurückgreifen, sollten sich dabei keine gravierenden Schwierigkeiten ergeben. Zum anderen könnte untersucht werden, ob sich nicht jeder XML-Prozess selbst als Webservice auffassen lässt. Ein Prozess bietet demnach einen Dienst im Netz an, der in Anspruch genommen werden kann, indem ihm ein SOAP-Dokument gesendet wird. Dieses wird vom Prozess verarbeitet und das resultierende XML-Dokument als Ergebnis, wiederum in Form einer SOAP-Nachricht, an den Aufrufer übermittelt. Möglicherweise sind aber noch andere Formen vorstellbar, beide Techniken miteinander zu verbinden.

Wie sich zeigt, gibt es eine Reihe interessanter, weiterführender Fragestellungen. Zum großen Teil ergaben sich diese erst während der Erarbeitung und Umsetzung der in dieser Arbeit vorgestellten Konzepte. Obwohl sie im Rahmen der Diplomarbeit nicht mehr vertiefend berücksichtigt werden konnten, lohnt es sich sicher, in den angesprochenen Bereichen weitergehende Überlegungen anzustellen.

Literatur und Quellen

[Aal97] W.M.P. van der Aalst:
The Application of Petri Nets to Workflow Management
The Journal of Circuits, Systems and Computers, 8(1): S. 21-66, 1998
http://wwwis.win.tue.nl/~wsinwa/jcsc/jcsc.html

[Ajm95] M. Ajmone Marsan et al.:
Modelling with Generalized Stochastic Petri Nets
John Wiley & Sons Ltd., 1995

[Alh98] Sinan Si Alhir:
UML in a Nutshell
O'Reilly, September 1998

[Alh99] Sinan Si Alhir:
Extending the Unified Modeling Language (UML)
Januar 1999
http://home.earthlink.net/~salhir/ExtendingTheUML.PDF

[Apa01] Apache Software Foundation:
Cocoon User Documentation
http://xml.apache.org/cocoon2/userdocs/index.html

[Ark01] Assaf Arkin:
Business Process Modeling Language (BPML)
Business Process Management Initiative, März 2001
http://www.bpmi.org/bpml-spec.esp

[Bac98] Jean Bacon:
Concurrent Systems
Addison Wesley Longman Ltd., 1998

[Bad01] Deutsche Bausparkasse Badenia AG:
Interner Anforderungskatalog für ein E-Business-Projekt des Unternehmens sowie diverse Quellcodes einer Realisierung

[Bau96] Bernd Baumgarten:
Petri-Netze, Grundlagen und Anwendungen
Spektrum, Akademischer Verlag, 1996

[Bet01] Urban Bettag:
Web-Services
Informatik Spektrum, Band 24, Heft 5, Oktober 2001
Springer-Verlag Berlin

[Bla00] M. Brian Blake:
WARP: An Agent-Based Process and Architecture for Workflow-Oriented Distributed Component Configuration
Proceedings of the 2000 International Conference on Artificial Intelligence (IC'AI2000), Las Vegas, Juni 2000
http://mason.gmu.edu/~mblake/icai2000.pdf

[Biz01] BizTalk.org:
BizTalk Framework (Web Site)
http://www.biztalk.org/home/framework.asp

[Boo99] Grady Booch, Jim Rumbaugh, Ivar Jacobson:
Das UML-Benutzerhandbuch
Addison Wesley Longman Ltd., 2. Auflage, 1999

[Böh95] Markus Böhm, Wolfgang Schulze:
Grundlagen von Workflow-Managementsystemen
Wissenschaftliche Beiträge zur Informatik
TU Dresden, 8 (1995) Heft 2, S.50-65
http://wwwdb.inf.tu-dresden.de/dokumente/ls-dokumente/wfgrundl.pdf

[Cow01] Danny Coward, Sun Microsystems, Inc.:
Java Servlet Specification Version 2.3
http://java.sun.com/aboutJava/communityprocess/first/jsr053/index.html

[CZ01] Computer Zeitung:
Modell senkt Folgekosten für eine einheitliche IT
Computer Zeitung Nr.32 vom 9. August 2001, S.14
Konradin-Verlag, Leinfelden

[Day91] Umeshwar Dayal et al.:
A Transactional Model for Long-Running Activities
Proceedings of the 17th International Conference on Very Large Data Bases, 1991
http://www.vldb.org/conf/1991/P113.PDF

[Dep00] R.Depke, M.Langham, B.Lütkemeier, S.Thöne:
Ein Konzept zur Spezifikation von XSL-Transformationen und dessen Anwendung bei Bankselbstbedienungssystemen
Tagungsband zu Net.Object Days 2000
Hrsg.: Net.ObjectDays-Forum

[DOM01] Philippe Le Hégaret et al.:
Document Object Model (DOM) Level 3 Core Specification, Version 1.0
W3C Working Draft, 13. September 2001
http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010913/

[Dum01] Marlon Dumas, Arthur H. M. ter Hofstede:
UML Activity Diagrams as a Workflow Specification Language
Proceedings at the 4th International Conference, Toronto, Canada, 2001
Lecture Notes of Computer Science 2185, S. 76ff
Springer-Verlag Berlin, 2001

[EbX01] UN/CEFACT, OASIS:
ebXML
http://www.ebxml.org/

[Esh01] Rik Eshuis, Roel Wieringa:
A Formal Semantics for UML Activity Diagrams
– Formalising Workflow Models
CTIT Technical Report 01-04
University of Twente, Februar 2001
http://www.ub.utwente.nl/webdocs/ctit/1/0000004e.pdf

[Gam96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Entwurfsmuster
– Elemente wiederverwendbarer objektorientierter Software
Addison Wesley, 1996

[Geo95] D. Georgakopoulos, M. Hornick, A. Sheth:
An Overview of Workflow Management:
From Process Modeling to Workflow Automation Infrastructure
Distributed and Parallel Databases, 3, S.119-153, 1995
Kluwer Academic Publishers, Boston

[Gru01] Volker Gruhn, Lothar Schöpe:
A Software Process for an Integrated Electronic Commerce Portal System
8th European Workshop on Software Process Technology (EWSPT 2001)
Lecture Notes of Computer Science 2077, S.90ff
Springer-Verlag Berlin, 2001
http://ls10-www.informatik.uni-dortmund.de/~schoepe/artikel/ewspt.pdf

[Gso01] Goldene Seiten Online:
Portale und virtuelle Marktplätze – Was ist ein Portal?
http://www.gso.at/service/handelsportale/wasisteinportal.html

[Hei00A] Hans-Ulrich Heiß:
Kommunikation zwischen Prozessen
Skriptum zur Vorlesung „Konzepte und Methoden der Systemsoftware“, 7-2
Universität Paderborn, 2000
http://www.uni-paderborn.de/cs/ag-heiss/lehre/kms/kms_7_4.pdf

[Hei00B] Hans-Ulrich Heiß:
Transaktionen in verteilten Systemen:
Skriptum zur Vorlesung „Verteilte Systeme I“, 6
Universität Paderborn, 2000
http://www.uni-paderborn.de/cs/ag-heiss/lehre/vs/vs_6_4.pdf

[Hen01] Hans-Thomas Hengl:
E-Business-Anwendungen bilden nur Inseln in der IT-Landschaft
Computer Zeitung Nr.36 vom 6. September 2001, S.9
Konradin-Verlag, Leinfelden

[HTML99] Dave Raggett et al.:
HTML 4.01 Specification
W3C World Wide Web Consortium Recommendation, 24. Dezember 1999
http://www.w3.org/TR/html401/

[ISO96] ISO:
Extended BNF
ISO/IEC 14977:1996(E)
International Organization for Standardization, Genf
http://www.dataip.co.uk/Reference/iso-14977.pdf

[Jec00] Mario Jeckle:
Entwurf von XML-Sprachen
Java Spektrum 6/00, SIGS Conferences GmbH, November 2000
http://www.jeckle.de/entwurfxml/entwurfxml.html

[Jin01] Sun Microsystems, Inc.:
Jini Network Technology, An Executive Overview
http://www.sun.com/jini/whitepapers/jini-execoverview.pdf

[Joh00] P. Johannesson, B. Wangler, und P. Jayaweera:
Application and Process Integration - Concepts, Issues, and Research Directions
Information Systems Engineering Symposium 2000
Hrsg.: Brinkkemper, Lindencrona und Sölvberg, Springer Verlag 2000
http://www.dsv.su.se/~perjons/fossilpaul2.pdf

[Kel92] G. Keller, M. Nüttgens, A.-W. Scheer:
Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“
Forschungsbericht des Instituts für Wirtschaftsinformatik
der Universität des Saarlandes, Heft 89, Januar 1992
http://www.iwi.uni-sb.de/iwi-hefte/heft089.zip

[Kel99] Gerhard Keller & Partner:
SAP R/3 prozeßorientiert anwenden
Iteratives Prozeß-Prototyping mit Ereignisgesteuerten Prozessketten und Knowledge Maps
Addison-Wesley Longman Verlag GmbH, 1999

[Kon96] Qinzheng Kong, Graham Chen:
Transactional Workflow For Telecommunication Service Management
IEEE/IFIP Network Operations and Management Symposium, 1996
http://www.citr.com.au/pdfs/96journ/96p3.pdf

[Lau00] Brett McLaughlin:
Java and XML
O'Reilly, Juni 2000
http://www.oreilly.com/catalog/javaxml/chapter/ch09.html

[Len01] Kirsten Lenz, Andreas Oberweis:
Modeling Interorganizational Workflows with XML Nets
Proceedings of the Hawaii International Conference On System Sciences, IEEE, 2001
ftp://kina.wiwi.uni-frankfurt.de/pub/publikationen/tagungen/t50.pdf

[Lüt00] Björn Lütkemeier, Sebastian Thöne:
Entwicklung eines Übersetzers von Nachrichtenaustauschformaten für Bankselbstbedienungssysteme in XML-Formate
Universität Paderborn, Fachbereich Informatik, Studienarbeit Mai 2000
http://home.vr-web.de/thoene/studium/studienarbeit.pdf

[Mak01] Makoto Murata, Dongwon Lee, Murali Mani:
Taxonomy of XML Schema Languages using Formal Language Theory
Extreme Markup Languages 2001, Montreal, Canada, August 2001
http://www.cobase.cs.ucla.edu/tech-docs/dongwon/mura0619.pdf

[Med93] R. Medina-Mora, T. Winograd, P. Flores:
Action Workflow as the Enterprise Integration Technology
Bulletin of the Technical Committee on Data Engineering, Vol. 16(2), 1993
IEEE Computer Society

[Meg00] Megginson Technologies:
SAX 2.0: The Simple API for XML, Mai 2000
http://www.megginson.com/SAX/

[Mic00] Microsoft Corporation:
BizTalk Orchestration –
A Technology for Orchestrating Business Interactions
White Paper, Juni 2000
http://www.microsoft.com/biztalk/techinfo/planning/2000/wp_orchestration.doc

[Mic01] Microsoft Corporation:
Microsoft BizTalk Server (Web Site)
http://www.microsoft.com/biztalk/default.asp

[MOF00] Object Management Group:
Meta Object Facility (MOF) Specification (Version 1.3, März 2000)
ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf

[Moh01] Stephen Mohr, Scott Woodgate:
Professional BizTalk
Wrox Press Inc., ISBN 1861003293, Januar 2001
http://www.wrox.com/books/samplechapters/3293/content.pdf

[Oes97] Bernd Oestereich:
Objektorientierte Geschäftsprozessmodellierung mit der UML
OBJECTspektrum, Ausgabe 2/98
Sigs Datacom GmbH, Troisdorf
http://www.oose.de/download/oogpm.pdf

[Omg99] Object Management Group:
UML Profile for Enterprise Distributed Object Computing
Request for Proposal, März 1999
ftp://ftp.omg.org/pub/docs/ad/99-03-10.pdf

[Omg00] Object Management Group:
OMG Unified Modeling Language Specification (Version 1.3, März 2000)
ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf

[Omg01] Object Management Group:
OMG Unified Modeling Language Specification
(Version 1.4, September 2001)
ftp://ftp.omg.org/pub/docs/formal/01-09-67.pdf

[Rit99] Peter Rittgen:
Modified EPCs and their formal semantics
Arbeitsberichte des Instituts für Wirtschaftsinformatik Nr.19,
Universität Koblenz-Landau, 1999
http://www.uni-koblenz.de/~rittgen/Nr19.pdf

[Rit00] Peter Rittgen:
Quo vadis EPK in ARIS ?
Ansätze zu syntaktischen Erweiterungen und einer formalen Semantik
WIRTSCHAFTSINFORMATIK 42 Heft 1
Vieweg Verlag, 2000
http://www.uni-koblenz.de/~rittgen/ZWI00.pdf

[Sch98] August-Wilhelm Scheer:
ARIS – Vom Geschäftsprozeß zum Anwendungssystem
Springer-Verlag Berlin, 1998

[SOAP01] Martin Gudgin et al.:
SOAP Version 1.2
W3C World Wide Web Consortium Working Draft, 9. Juli 2001
http://www.w3.org/TR/2001/WD-soap12-20010709/

[Sun01] S&N AG:
sunShine
http://www.sundn.de/produkte/sunshine/sunshine.htm

[Tha00] Satish Thatte:
XLANG – Web Services For Business Process Design
Microsoft Corporation, 2001
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[XMI00] Object Management Group:
OMG XML Metadata Interchange (XMI) Specification
(Version 1.1, November 2000)
ftp://ftp.omg.org/pub/docs/formal/00-11-02.pdf

[XML00] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler:
Extensible Markup Language (XML) 1.0 (Second Edition)
W3C World Wide Web Consortium Recommendation, 6. Oktober 2000
http://www.w3.org/TR/2000/REC-xml-20001006

[XPath99] James Clark, Steve DeRose:
XML Path Language (XPath) Version 1.0
W3C World Wide Web Consortium Recommendation, 16. November 1999
http://www.w3.org/TR/1999/REC-xpath-19991116

[XQL98] Jonathan Robie et al.:
XML Query Language (XQL)
W3C World Wide Web Consortium, September 1998
http://www.w3.org/TandS/QL/QL98/pp/xql.html

[XQuery01] Don Chamberlin et al.:
XQuery 1.0: An XML Query Language
W3C World Wide Web Consortium Working Draft, 07. Jun. 2001
http://www.w3.org/TR/xquery/

[XSch01] David C. Fallside:
XML Schema Part 0 : Primer
W3C World Wide Web Consortium Recommendation, 02. Mai 2001
http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

[XSLT99] James Clark:
XSL Transformations (XSLT) Version 1.0
W3C World Wide Web Consortium Recommendation, 16. Nov. 1999
http://www.w3.org/TR/1999/REC-xslt-19991116

[Wen00] Rüdiger Wenski:
Eine objektorientierte Systemkomponente zur Workflow-Modellierung und –Ausführung unter besonderer Berücksichtigung der Telekooperation
Diss., Heinz-Nixdorf Institut Paderborn, HNI-Verlagsschriftenreihe, 2000

[Wes01] Berthold Wesseler:
Integration-Server automatisieren Geschäftsprozesse und sparen Zeit
Computer Zeitung Nr.32 vom 9. August 2001, S.14
Konradin-Verlag, Leinfelden

[Wet93] Horst Wettstein:
Systemarchitektur
Carl Hanser Verlag München Wien, 1993

[Wie01] Rik Eshuis, Roel Wieringa:
An Execution Algorithm for UML Activity Graphs
Proceedings at the 4th International Conference, Toronto, Canada, 2001
Lecture Notes of Computer Science 2185, S. 47ff
Springer-Verlag Berlin, 2001

[Wmc95] Workflow Management Coalition:
The Workflow Reference Model
WfMC-TC-1003, Version 1.1, Januar 1995
http://www.wfmc.org/standards/docs/tc003v11.pdf

[Wmc99] Workflow Management Coalition:
Terminology & Glossary
WfMC-TC-1011, Version 3, Februar 1999
Spezifikation der Workflow Management Coalition
http://www.wfmc.org/standards/docs/TC-1011_term_glossary_v3.pdf

[WSDL01] Erik Christensen et al.:
Web Services Description Language (WSDL) 1.1
W3C World Wide Web Consortium Note, 15. Mai 2000
http://www.w3.org/TR/2001/NOTE-wsdl-20010315

Anmerkungen:

Bei den genannten Firmen- und Produktnamen handelt es sich zum Teil um eingetragene Warenzeichen. Im Sinne der besseren Lesbarkeit ist dies jedoch nicht explizit gekennzeichnet.

Alle angegebenen Internetadressen entsprechen dem Stand von Dezember 2001. Zwischenzeitliche Änderungen sind nicht auszuschließen.

A Anhang

A.1 Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
ARIS	Architektur integrierter Informationssysteme
B2B	Business to Business
B2C	Business to Consumer
BPMI	Business Process Management Initiative
BPML	Business Process Modeling Language
CASE	Computer Aided Software Engineering
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CSCW	Computer Supported Cooperative Work
DB	Database
DOM	Document Object Model
DTD	Document Type Definition
EAI	Enterprise Application Integration
ebXML	Electronic Business XML
EDI	Electronic Data Interchange
EPK	Ereignisgesteuerte Prozesskette
ERP	Enterprise Resource Planning
GWMA	Global Workflow Manager Agent
GXSL	Graphical XML Schema Definition Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
MOF	Meta Object Facility
MSL	Model Schema Language
OCL	Object Constraint Language
OMG	Object Management Group
PDF	Portable Document Format
PML	Process Markup Language
RFC	Remote Function Call
RMA	Role Manager Agent
RMI	Remote Method Invocation
SAP	Systeme, Anwendungen und Produkte in der Datenverarbeitung
SAX	Simple API for XML
SMA	Site Manager Agent
SMTP	Simple Mail Transport Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WARP	Workflow Automation through Agent-based Reflective Processes

WFMS	Workflow Management System
WMA	Workflow Manager Agent
WML	Wireless Markup Language
WSDL	Web Service Description Language
XML	Extensible Markup Language
XPath	XML Path Language
XQL	XML Query Language
XSL	Extensible Stylesheet Language

A.2 Inhalt der CD

Der Diplomarbeit ist ein CD-ROM-Datenträger beigefügt, auf dem sich das sunFlow-System einschließlich einiger Beispieldaten sowie Materialien zur Dokumentation befinden.

Die Verzeichnisse haben folgenden Inhalt:

/Dokumentation	
/Diplomarbeit.doc	Word-Version dieser Arbeit
/Diplomarbeit.pdf	PDF-Version dieser Arbeit
/Präsentation.ppt	Präsentation der Ergebnisse
/sunFlow	
/lib	
/fujaba-runtime.jar	Hilfsbibliothek zur Implementierung von Assoziationen zwischen Klassen
/xerces.jar	Der XML-Parser *Xerces* der Apache-Organisation
/xalan.jar	Der XSL-Prozessor *Xalan* der Apache-Organisation
/source/	Quellcode des sunFlow-Systems
/javadoc/	Quellcode-Dokumentation als JavaDoc
/sunFlow.jar	Archiv mit den lauffähigen Klassendateien
/sunflow.properties	Property-Datei
/PML-Spezifikation	
/PMLSchema.xsd	XML-Schema für die Sprache PML
/Beispieldaten/	Einige Beispieldaten, die mit dem sunFlow-System verarbeitet werden können

A.3 Autorenzuordnung

1 Einleitung (Thöne)

2 Grundlagen des Workflow-Managements (Thöne)

3 Anforderungsanalyse
- 3.1 Fallstudie: E-Business-Plattform eines großen Finanzunternehmens (Thöne)
- 3.2 Anforderungen an Workflow-Systeme im E-Business (Lütkemeier)
- 3.3 Anforderungen an Workflow-Modellierungssprachen (Lütkemeier)
- 3.4 Zielsetzungen dieser Arbeit (Thöne)

4 Evaluierung vorhandener Workflow-Systeme (Lütkemeier)
bis auf 4.1 Microsoft BizTalk (Thöne)

5 Evaluierung vorhandener Workflow-Modellierungssprachen (Lütkemeier)
bis auf 5.2 Ereignisgesteuerte Prozessketten (Thöne)

6 XML-Prozesse und Prozessdiagramme
- 6.1 Konzeption von XML-Prozessen (Thöne)
- 6.2 Modellierung mit Prozessdiagrammen (Lütkemeier)
- 6.3 Modellierung der Dokumenttypen (Thöne)
- 6.4 Bewertung (Lütkemeier)

7 Metamodell für Prozessdiagramme
- 7.1 Das UML-Metamodell (Lütkemeier)
- 7.2 UML-Erweiterungsmechanismen (Thöne)
- 7.3 UML-Profil für Prozessdiagramme (Thöne)
- 7.4 Implementierung des erweiterten Metamodells (Thöne)
- 7.5 Validierung von Prozessdiagrammen (Lütkemeier)
- 7.6 Automatische Portberechnung (Thöne)

8 Beschreibung von Prozessen in XML
- 8.1 Anforderungen an Prozessbeschreibungssprachen (Thöne)
- 8.2 Evaluierung vorhandener Prozessbeschreibungssprachen (Thöne)
  bis auf 8.2.1 Beschreibung von Prozessen mit XLANG (Lütkemeier)
- 8.3 Process Markup Language (PML) (Lütkemeier)
- 8.4 Werkzeuge für die PML-Verarbeitung (Lütkemeier)

9 Modellierungswerkzeug für Prozessdiagramme (Lütkemeier)

10 Ausführung von XML-Prozessen (Lütkemeier)
bis auf 10.3 Fehlerbehandlung (Thöne)

11 Evaluierung anhand einer Fallstudie (Thöne)

12 Schlussbetrachtungen
- 12.1 Zusammenfassung (Thöne)
- 12.2 Ausblick (Lütkemeier)